## Contents

1 Support .......................................................... 1

2 First steps ...................................................... 3

3 The model layer ................................................ 5

4 The view layer .................................................. 7

5 The development process ..................................... 9

6 Contents ......................................................... 11
   6.1 Using trytond ................................................. 11
   6.2 API Reference ............................................... 51

7 Indices, glossary and tables .................................. 85

Python Module Index ............................................. 87

Index .................................................................. 89
Support

- Looking for specific information? Try the genindex, modindex.
- Search information in the mailing list.
- Ask a question in the #tryton IRC channel, or search the IRC logs.
- Report issues with tryond on issue tracker.
First steps

- **Installation**: Installation | Configuration | Setup a database | Start the server
CHAPTER 3

The model layer

- Models: Model syntax | Field types | Domain syntax | Access rights | Triggers
CHAPTER 4

The view layer

• Views: View types | Extension
• Reports: Report definition
The development process

• Modules Module definition
6.1 Using trytond

Introduction to all the key parts of trytond:

6.1.1 How to install Tryton

Prerequisites

- Python 2.7 or later (http://www.python.org/)
- Werkzeug (http://werkzeug.pocoo.org/)
- wrapt (https://github.com/GrahamDumpleton/wrapt)
- lxml 2.0 or later (http://lxml.de/)
- relatorio 0.7.0 or later (http://relatorio.tryton.org/)
- Genshi (http://genshi.edgewall.org/)
- python-dateutil (http://labix.org/python-dateutil)
- polib (https://bitbucket.org/izi/polib/wiki/Home)
- python-sql 0.4 or later (http://code.google.com/p/python-sql/)
- Optional: psycopg 2.5.0 or later (http://www.initd.org/)
- Optional: psycopg2cffi 2.5.0 or later (http://github.com/chtd/psycopg2cffi)
- Optional: MySQL-python (http://sourceforge.net/projects/mysql-python/)
- Optional: pydot (http://code.google.com/p/pydot/)
- Optional: unoconv http://dag.wieers.com/home-made/unoconv/)
- Optional: sphinx (http://sphinxc.pocoo.org/)
• Optional: cdecimal (http://www.bytereef.org/mpdecimal/index.html)
• Optional: python-Levenshtein (http://github.com/miohtama/python-Levenshtein)
• Optional: bcrypt (https://github.com/pyca/bcrypt)
• Optional: html2text (https://pypi.python.org/pypi/html2text)
• Optional: mock (http://www.voidspace.org.uk/python/mock/)

Install Tryton

There are three easy options to install Tryton:

• Install the version provided by your operating system distribution. This is the quickest and recommended option for those who has operating system that distributes Tryton.

• Install an official release. Once you’ve downloaded and unpacked a trytond source release, enter the directory where the archive was unpacked, and run: python setup.py install

For advanced options, please refer to the easy_install and/or the distutils documentation:

– http://docs.python.org/inst/inst.html

• Without installation, just run bin/trytond from where the archive was unpacked.

Warning: Note that you may need administrator/root privileges for this step, as this command will by default attempt to install trytond to the Python site-packages directory on your system.

6.1.2 Configuration file for Tryton

The configuration file controls some aspects of the behavior of Tryton. The file uses a simple ini-file format. It consists of sections, led by a [section] header and followed by name = value entries:

```
[database]
url = postgresql://user:password@localhost/
path = /var/lib/trytond
```

For more information see ConfigParser.

Sections

This section describes the different main sections that may appear in a Tryton configuration file, the purpose of each section, its possible keys, and their possible values. Some modules could request the usage of other sections for which the guideline asks them to be named like their module.

web

Defines the behavior of the web interface.
listen

Defines the couple of host (or IP address) and port number separated by a colon to listen on.
Default `localhost:8000`

hostname

Defines the hostname.

root

Defines the root path served by `GET` requests.
Default: Under the `www` directory of user’s home running `tryond`.

database

Defines how the database is managed.

uri

Contains the URI to connect to the SQL database. The URI follows the RFC-3986. The typical form is:
```python
database://username:password@host:port/
```
Default: `sqlite://`

The available databases are:

**PostgreSQL**

`pyscopg2` supports two type of connections:
- TCP/IP connection: `postgresql://user:password@localhost:5432/`
- Unix domain connection: `postgresql://username:password@/`

**SQLite**

The only possible URI is: `sqlite://`

**MySQL**

Same as for PostgreSQL.
path

The directory where Tryton stores files and so the user running tryond must have write access on this directory. Default: The db folder under the user home directory running tryond.

list

A boolean value to list available databases. Default: True

retry

The number of retries when a database operational error occurs during a request. Default: 5

language

The main language of the database that will be used for storage in the main table for translations. Default: en

request

max_size

The maximum size in bytes of unauthenticated request (zero means no limit). Default: 2MB

max_size_authenticated

The maximum size in bytes of an authenticated request (zero means no limit). Default: 2GB

cache

Defines size of various cache.

model

The number of different model kept in the cache per transaction. Default: 200
record

The number of record loaded kept in the cache of the list. It can be changed locally using the `_record_cache_size` key in `Transaction.context`.

Default: 2000

field

The number of field to load with an `eager Field.loading`.

Default: 100

table

This section allows to override the default generated table name for a `ModelSQL`. The main goal is to bypass limitation on the name length of the database backend. For example:

```
[table]
account.invoice.line = acc_inv_line
account.invoice.tax = acc_inv_tax
```

ssl

Activates SSL on all network protocols.

**Note:** SSL is activated by defining `privatekey`. Please refer to SSL-CERT on how to use private keys and certificates.

privatekey

The path to the private key.

certificate

The path to the certificate.

e-mail

uri

The SMTP-URL to connect to the SMTP server which is extended to support SSL and STARTTLS. The available protocols are:

- `smtp`: simple SMTP
- `smtp+tls`: SMTP with STARTTLS
- `smtps`: SMTP with SSL
The uri accepts the following additional parameters:

- `local_hostname`: used as FQDN of the local host in the HELO/EHLO commands, if omitted it will use the value of `socket.getfqdn()`.
- `timeout`: A number of seconds used as timeout for blocking operations. A `socket.timeout` will be raised when exceeded. If omitted the default timeout will be used.

Default: `smtp://localhost:25`

**from**

Defines the default *From* address for emails sent by Tryton.

**session**

**authentications**

A comma separated list of login methods to use to authenticate the user. By default, Tryton supports only the `password` method which compare the password entered by the user against a stored hash. But other modules can define new methods (please refers to their documentation). The methods are tested following the order of the list.

Default: `password`

**timeout**

The time in seconds until a session expires.

Default: `600`

**max_attempt**

The maximum authentication attempt before the server answers unconditionally *Too Many Requests* for any other attempts. The counting is done on all attempts over a period of `timeout`.

Default: `5`

**password**

**length**

The minimal length required for the user password.

Default: `8`

**forbidden**

The path to a file containing one forbidden password per line.
entropy

The ratio of non repeated characters for the user password.
Default: 0.75

report

unoconv

The parameters for unoconv.
Default: pipe.name=trytond;urp;StarOffice.ComponentContext

attachment

Defines how to store the attachments

filestore

A boolean value to store attachment in the FileStore.
Default: True

store_prefix

The prefix to use with the FileStore.
Default: None

6.1.3 How to setup a database

The database section of the configuration must be set before starting.

Create a database

Depending of the database backend choosen, you must create a database (see the documentation of the choosen backend). The user running trytond must be granted the privilege to create tables. For backend that has the option, the encoding of the database must be set to UTF-8.

Initialize a database

A database can be initialized using this command line:

```
trytond-admin -c <config file> -d <database name> --all
```

At the end of the process, trytond-admin will ask to set the password for the admin user.
Update a database

To upgrade to a new series, the command line is:

```
trytond-admin -c <config file> -d <database name> --all
```

**Warning:** Prior to upgrade see if there is no manual action to take on the migration topic.

To activate a new language on an existing database, the command line is:

```
trytond-admin -c <config file> -d <database name> --all -l <language code>
```

Once activated, the language appears in the user preferences.

### 6.1.4 Logging configuration

Without any configuration, trytond writes ERROR messages to standard output. With the verbose flag set, it writes INFO message. And with the verbose and development flags set, it write DEBUG message.

Logs can be configured using a configparser-format file. The filename can be specified using trytond logconf parameter.

**Example**

This example allows to write INFO messages on standard output and on a disk log file rotated every day.

```ini
[formatters]
keys=simple

[handlers]
keys=rotate,console

[loggers]
keys=root

[formatter_simple]
format=%(asctime)s [%(levelname)s:%(name)s] %(message)s
datefmt=%a %b %d %H:%M:%S %Y

[handler_rotate]
class=handlers.TimedRotatingFileHandler
args=('~/tmp/tryton.log', 'D', 1, 30)
formatter=simple

[handler_console]
class=StreamHandler
formatter=simple
args=(sys.stdout,)

[logger_root]
level=INFO
handlers=rotate,console
```
6.1.5 How to start the server

Web service

You can start the default web server bundled in Tryton with this command line:

```
trytond -c <config file>
```

The server will wait for client connections on the interface defined in the web section of the configuration.

**Note:** When using multiple config files the order is important as last entered files will override the items of first files

WSGI server

If you prefer to run Tryton inside your own WSGI server instead of the simple server of Werkzeug, you can use the application `trytond.application.app`. Following environment variables can be set:

- `TRYTOND_CONFIG`: Point to configuration file.
- `TRYTOND_LOGGING_CONFIG`: Point to logging file.
- `TRYTOND_DATABASE_NAMES`: A list of database names in CSV format, using python default dialect.

**Warning:** You must manage to serve the static files from the web root.

Cron service

If you want to run some scheduled actions, you must also run the cron server with this command line:

```
trytond-cron -c <config file> -d <database>
```

The server will wake up every minutes and perform the scheduled actions defined in the database.

Services options

You will find more options for those services by using `-help` arguments.

6.1.6 Models

A model represents a single business logic or concept. It contains fields and defines the behaviors of the record. Most of the time, each model stores records in a single database table.

The basics:

- Each model is a Python class that subclasses one of `trytond.model.model.Model`.
- *Fields* are defined as model attributes.
- Tryton generates the table definitions
- Tryton provides an API following the active record pattern to access the records.
Example

This example defines a `Party` model which has a `name` and a `code` fields:

```python
from trytond.model import ModelView, ModelSQL, fields

class Party(ModelSQL, ModelView):
    __name__ = "party.party"
    name = fields.Char('Name')
    code = fields.Char('Code')
```

The class must be registered in the *Pool* by the `register()` method of the *module*. Model classes are essentially data mappers to records and Model instances are records.

Model attributes define meta-information of the model. They are class attributes starting with an underscore. Some model properties are instance attributes allowing to update them at other places in the framework.

### 6.1.7 Default value of fields

When a record is created, each field, which doesn’t have a value specified, is set with the default value if exists.

The following class method:

```python
Model.default_<field name>()
```

Return the default value for `field name`.

This example defines an `Item` model which has a default `since`:

```python
import datetime
from trytond.model import ModelView, ModelSQL, fields

class Item(ModelSQL, ModelView):
    __name__ = 'item'
    since = fields.Date('since')

    @classmethod
    def default_since(cls):
        return datetime.date.today()
```

See also method `Model.default_get: default_get`

### 6.1.8 on_change of fields

Tryton allows developers to define methods that can be called once a field’s value has changed by the user this is the `on_change` method. The method has the following name:

```python
Model.on_change_<field name>
```

This is an instance method, an instance of `Model` will be created by using the values from the form’s fields specified by the `on_change` list defined on the field. Any change made on the instance will be pushed back to the client-side record.
There is also a way to define a method that must update a field whenever any field from a predefined list is modified. This list is defined by the `on_change_with` attribute of the field. The method that will be called has the following name:

```python
Model.on_change_with_<field_name>
```

Just like for the classic `on_change`, an instance of `Model` will be created by using the values entered in the form’s fields specified by the `on_change_with` attribute. The method must return the new value of the field to push back to the client-side record.

### 6.1.9 Domain

Domains represent a set of records. A domain is a list of none or more clauses. A clause is a condition, which returns true or false. A record belongs to a domain, when the final result of the list of clauses returns true.

**Syntax**

The definition of a simple domain with one clause is represented by this pattern:

```python
domain = [(<field name>, <operator>, <operand>)]
```

- `<field name>` is the name of a `trytond.model.fields` or a `pyson` statement, that evaluates to a string.
  A field of type `trytond.model.fields.Many2One` or `trytond.model.fields.Many2Many` or `trytond.model.fields.One2Many` or `trytond.model.fields.One2One` or `trytond.model.fields.Reference` can be dereferenced to related models. This is illustrated by the following example:

```python
domain = [('country.name', '=', 'Japan')]
```

- The number of `dots` in a clause is not limited.

**Warning**: For `trytond.model.fields.Reference`, an extra ending clause is needed to define the target model to join, for example:

```python
domain = [('origin.party.name', '=', 'John Doe', 'sale.sale')]
```

- `operator` is an operator out of `Domain Operators` or a `pyson` statement, that evaluates to a domain operator string.
- `operand` is an operand or a `pyson` statement. The type of operand depends on the kind of `<field name>`.

The definition of an empty domain is:

```python
domain = []
```

An empty domain without clauses will always return all `active` records. A record is active, when its appropriate Model contains a `Boolean` field with name `active`, and set to true. When the appropriate Model does not contain a `Boolean` field with name `active` all records are returned.

A domain can be setup as a combination of clauses, like shown in this pattern:

```python
domain = [,
  ('field name1', 'operator1', 'operand1'),
  ('field name2', 'operator2', 'operand2'),
  ('field name3', 'operator3', 'operand3'),
]
```
The single clauses are implicitly combined with a logical AND operation. In the domain syntax it is possible to provide explicitly the combination operation of the clauses. These operations can be AND or OR. This is illustrated by the following pattern:

```plaintext
domain = [ 'OR', [
    ('field name1', 'operator1', 'operand1'),
    ('field name2', 'operator2', 'operand2'),
], [
    ('field name3', 'operator3', 'operand3'),
],
```

Here the domain is evaluated like this: ((clause1 AND clause2) OR clause3). Please note that the AND operation is implicit assumed when no operator is given. While the OR operation must be given explicitly. The former pattern is equivalent to the following completely explicit domain definition:

```plaintext
domain = [ 'OR', [
    'AND', [
        ('field name1', 'operator1', 'operand1'),
    ], [
        ('field name2', 'operator2', 'operand2'),
    ],
], [
    ('field name3', 'operator3', 'operand3'),
],
```

Obviously the use of the implicit AND operation makes the code more readable.

**Domain Operators**

The following operators are allowed in the domain syntax. `<field name>`, `<operator>` and `<operand>` are dereferenced to their values. The description of each operator follows this pattern, unless otherwise noted:

```
(<field name>, <operator>, <operand>)
```

### =

Is a parity operator. Returns true when `<field name>` equals to `<operand>`. 

### !=

Is an imparity operator. It is the negation of the `=` operator. 

### like

Is a pattern matching operator. Returns true when `<field name>` is contained in the pattern represented by `<operand>`. In `<operand>` an underscore (_) matches any single character, a percent sign (%) matches any string with zero or more characters. To use _ or % as literal, use the backslash \\ to escape them. All matching is case sensitive.
not like

Is a pattern matching operator. It is the negation of the like operator.

ilike

Is a pattern matching operator. The same use as like operator, but matching is case insensitive.

not ilike

Is a pattern matching operator. The negation of the ilike operator.

in

Is a list member operator. Returns true when <field name> is in <operand> list.

not in

Is a list non-member operator. The negation of the in operator.

<

Is a less than operator. Returns true for type string of <field name> when <field name> is alphabetically sorted before <operand>.

Returns true for type number of <field name> when <field name> is less than <operand>.

>

Is a greater than operator. Returns true for type string of <field name> when <field name> is alphabetically sorted after <operand>.

Returns true for type number of <field name> when <field name> is greater <operand>.

<=

Is a less than or equal operator. Returns the same as using the < operator, but also returns true when <field name> is equal to <operand>.

>=

Is a greater than or equal operator. Returns the same as using the > operator, but also returns true when <field name> is equal to <operand>.
child_of

Is a parent child comparison operator. Returns true for records that are a child of `<operand>`. `<operand>` is a list of ids and `<field name>` must be a many2one or a many2many. In case `<field name>` is not linked to itself, the clause pattern extends to:

```
[{<field name>, ['child_of'|'not_child_of'], <operand>, <parent field>}]`
```

Where `<parent field>` is the name of the field constituting the many2one on the target model.

not child_of

Is a parent child comparison operator. It is the negation of the `child_of` operator.

parent_of

Is a parent child comparison operator. It is the same as `child_of` operator but if `<field name>` is a parent of `<operand>`.

not parent_of

Is a parent child comparison operator. It is the negation of this `parent_of` operator.

where

Is a `trytond.model.fields.One2Many / trytond.model.fields.Many2Many` domain operator. It returns true for every row of the target model that match the domain specified as `<operand>`.

not where

Is a `trytond.model.fields.One2Many / trytond.model.fields.Many2Many` domain operator. It returns true for every row of the target model that does not match the domain specified as `<operand>`.

6.1.10 PYSON

PYSON is the PYthon Statement and Object Notation. It is a lightweight domain specific language for the general representation of statements. PYSON is used to encode statements which can be evaluated in different programming languages, serving for the communication between trytond and any third party software. A PYSON parser can easily be implemented in other programming languages. So third party softwares do not need to depend on Python to be able to fully communicate with the Tryton server.

PYSON is a deterministic algorithm which will always succeed to evaluate statements. There is a default behavior for unknown values. It is statically typed and checked on instantiation.

There is also a reference documentation of the API.
Syntax

The syntax of a PYSON statement follows this pattern:

\[
\text{Statement}(\text{argument1[, argument2[, ...]]})
\]

where arguments can be another statement or a value. The evaluation direction is inside out, deepest first.

PYSON Examples

Given the PYSON statement:

\[
\text{Eval('active\_id', -1)}
\]

Eval() checks the evaluation context for the variable active_id and returns its value or -1 if not defined. A similar expression in Python looks like this:

\[
'active\_id' \text{ in locals()} \text{ and active\_id or -1}
\]

Given the PYSON statement:

\[
\text{Not(Bool(Eval('active')))}
\]

Eval() checks the evaluation context for a variable active and returns its value to Bool() or '' if not defined. Bool() returns the corresponding boolean value of the former result to Not(). Not() returns the boolean negation of the previous result. A similar expression in Python looks like this:

\[
'active' \text{ in locals()} \text{ and active == False}
\]

Given the PYSON statement:

\[
\text{Or(Not(Equal(Eval('state'), 'draft')), Bool(Eval('lines')))}
\]

In this example are the results of two partial expressions Not(Equal(Eval('state'), 'draft')) and Bool(Eval('lines')) evaluated by a logical OR operator. The first expression part is evaluated as follow: When the value of Eval('state') is equal to the string 'draft' then return true, else false. Not() negates the former result. A similar expression in Python looks like this:

\[
'states' \text{ in locals()} \text{ and 'lines' in locals()} \text{ \}
\text{ and state != 'draft' or bool(lines)}
\]

Given the PYSON statement:

\[
\text{If(In('company', Eval('context', {})), '=', '!=')}
\]

In this example the result is determined by an if-then-else condition. In('company', Eval('context', {})) is evaluated like this: When the key 'company' is in the dictionary context, returns true, otherwise false. If() evaluates the former result and returns the string '=' if the result is true, otherwise returns the string '!='. A similar expression in Python looks like this:

\[
'context' \text{ in locals()} \text{ and isinstance(context, dict)} \text{ \}
\text{ and 'company' in context and '=' or '!='}
\]

Given the PYSON statement:
Eval() checks the evaluation context for a variable context if defined, return the variable context, otherwise return an empty dictionary {}. Get() checks the former resulting dictionary and returns the value of the key 'company', otherwise it returns the number 0. A similar expression in Python looks like this:

```python
'context' in locals() and context.get('company', 0) or 0
```

### 6.1.11 Access Rights

There are 5 levels of access rights: model, actions, field, button and record. Every access right is based on the groups of the user. The model and field access rights are checked for every RPC call for which `tryond.rpc.RPC.check_access` is set. The others are always enforced.

#### Model Access

They are defined by records of `ir.model.access` which define for each couple of model and group, the read, write, create and delete permission. If any group of the user has the permission activated, then the user is granted this permission.

#### Actions Access

Each action define a list of groups that are allowed to use it. There is a special case for `wizard` for which the read access on the model is also checked and also the write access if there is no groups linked.

#### Field Access

Same as for model access but applied on the field. It uses records of `ir.model.field.access`.

#### Button

For each button of a model the records of `ir.model.button` define the list of groups that are allowed to call it.

#### Button Rule

The `ir.model.button` could contain a list of rules which define how much different users must click on the button. Each rule must be passed to actually trigger the action. The counter can be reset when another defined button is clicked.

#### Record Rule

They are defined by records of `ir.rule.group` which contains a list of `ir.rule` domain to which the rule applies. The group are selected by groups or users. The access is granted for a record:

- if the user is in at least one group that has the permission activated,
- or if the user is in no group by there is a default group with the permission,
- or if there is a global group with the permission.
6.1.12 Triggers

Triggers allow to define methods of `trytond.model.model.Model` that are called when one of those events happen to a record:

- On Creation
- On Modification
- On Deletions
- On Time: When a condition changes over time.

The method signature is:

```
<method name>(cls, records, trigger)
```

Where `records` is the list of records that triggered the event and `trigger` is the `ir.trigger` instance which is triggered.

Triggers are defined by records of `ir.trigger`. Each record must define a pyson condition which will be evaluated when the event occurs. Only those records for which the condition is evaluated to true will be processed by the trigger with the exception of modification triggers which will only process the records for which the condition is evaluated to false before and evaluated to true after the modification.

6.1.13 Actions

Actions are used to describe specific behaviors in the client.

There are four types of actions:

- Report
- Window
- Wizard
- URL

Keyword

Keywords define where to display the action in the client.

There are five places:

- Open tree (`tree_open`)
- Print form (`form_print`)
- Action form (`form_action`)
- Form relate (`form_relate`)
- Open Graph (`graph_open`)

Report

Window

The window action describe how to create a new tab in the client.
View

Domain

The window action could have a list of domains which could be activated on the view. The boolean field count indicates if the client must display the number of records for this domain.

**Warning:** The counting option must be activated only on domains which have not too much records otherwise it may overload the database.

Wizard

URL

6.1.14 Views

The views are used to display records of an `ModelView` to the user.

In Tryton, `ModelView` can have several views. An *action* opens a window and defines which view to show.

The views are built from XML that is stored in the `view` directory of the module or in the databases thanks to the model `ir.ui.view`.

So generally, they are defined in xml files with this kind of xml where name is the name of the XML file in the `view` directory:

```xml
<record model="ir.ui.view" id="view_id">
    <field name="model">model name</field>
    <field name="type">type name</field>
    <!--field name="inherit" ref="inherit_view_id"-->
    <!--field name="field_childs">field name</field-->
    <field name="name">view_name</field>
</record>
```

There is three types of views:

- Form
- Tree
- Graph
- Board
- Calendar

Form view

The RNG describing the xml of a form view is stored in `trytond/ir/ui/form.rng`. There is also a RNC in `trytond/ir/ui/form.rnc`.

A form view is used to display one record of an object.

Elements of the view are put on the screen following the rules:

- Elements are placed on the screen from left to right, from top to bottom, according to the order of the xml.
• The screen composed of a table with a fixed number of columns and enough rows to handle all elements.
• Elements take one or more columns when they are put in the table. If there are not enough free columns on the current row, the elements are put at the beginning of the next row.

**XML description**

List of attributes shared by many form elements:

• *id*: A unique identifier for the tag if there is no name attribute.
• *yexpand*: A boolean to specify if the label should expand to take up any extra vertical space.
• *yfill*: A boolean to specify if the label should fill the vertical space allocated to it in the table cell.
• *yalign*: The vertical alignment, from 0.0 to 1.0.
• *xexpand*: The same as yexpand but for horizontal space.
• *xfill*: The same as yfill but for horizontal space.
• *xalign*: The horizontal alignment, from 0.0 to 1.0.
• *colspan*: The number of columns the widget must take in the table.
• *col*: The number of columns the container must have.
  A negative value (or zero) will remove the constraint on the number of columns.
  The default value is 4.
• *states*: A string of *PYSON statement* that will be evaluated with the values of the current record.
  It must return a dictionary where keys can be:
  – *invisible*: If true, the widget will be hidden.
  – *required*: If true, the field will be required.
  – *readonly*: If true, the field will be readonly.
  – *icon*: Only for button, it must return the icon name to use or False.
  – *pre_validate*: Only for button, it contains a domain to apply on the record before calling the button.
• *help*: The string that will be displayed when the cursor hovers over the widget.
• *pre_validate*: A boolean only for fields *trytond.model.fields.One2Many* to specify if the client must pre-validate the records using *trytond.model.Model.pre_validate()*.
• *completion*: A boolean only for fields *trytond.model.fields.Many2One, trytond.model.fields.Many2Many* and *trytond.model.fields.One2Many* to specify if the client must auto-complete the field. The default value is True.
• *factor*: A factor to apply on fields *trytond.model.fields.Integer, trytond.model.fields.Float* and *trytond.model.fields.Numeric* to display on the widget. The default value is 1.

**form**

Each form view must start with this tag.
• on_write: The name of a method on the Model of the view that will be called when a record is saved. The method must return a list of record ids that the client must reload if they are already loaded. The function must have this syntax:

```python
on_write(self, ids)
```

**Note:** The method must be registered in `trytond.model.Model.__rpc__`.

• col: see in `common-attributes-col`.

• cursor: The name of the field that must have the cursor by default.

**label**

Display static string.

• string: The string that will be displayed in the label.

• name: The name of the field whose description will be used for string. Except if `string` is set, it will use this value and the value of the field if `string` is empty.

• id: see `common-attributes-id`.

• yexpand: see in `common-attributes-yexpand`.

• yfill: see in `common-attributes-yfill`.

• yalign: see in `common-attributes-yalign`.

• xexpand: see in `common-attributes-xexpand`.

• xfill: see in `common-attributes-xfill`.

• xalign: see in `common-attributes-xalign`.

• colspan: see in `common-attributes-colspan`.

• states: see in `common-attributes-states`.

• help: see in `common-attributes-help`.

• Requires that either `id` or `name` is defined.

**field**

Display a field of the object with the value of the current record.

• name: The name of the field.

• string: The string that will be displayed for the widget.

• widget: The widget that must be used instead of the default one.

• help: The string that will be displayed when the cursor stays over the widget.

• width: The minimum width the widget should request, or -1 to unset.

• height: The minimum height the widget should request, or -1 to unset.

• readonly: Boolean to set the field readonly.
• **mode**: Only for One2Many fields: it is a comma separated list, that specifies the order of the view used to display the relation. (Example: `tree,form`)

• **view_ids**: A comma separated list that specifies the view ids used to display the relation.

• **product**: Only for One2Many fields, a comma separated list of target field name used to create records from the cartesian product.

• **completion**: Only for Many2One fields, it is a boolean to set the completion of the field.

• **invisible**: The field will not be displayed, but it will fill cells in the table.

• **filename_visible**: Only for Binary fields, boolean that enables the display of the filename.

• **toolbar**: Only for Rich Text widget, boolean that enables the display of the Rich Text toolbar. The default value is 1.

• **yexpand**: see in `common-attributes-yexpand`.

• **yfill**: see in `common-attributes-yfill`.

• **xexpand**: see in `common-attributes-xexpand`.

• **xfill**: see in `common-attributes-xfill`.

• **colspan**: see in `common-attributes-colspan`.

• **help**: see in `common-attributes-help`.

• **pre_validate**: see in `common-attributes-pre_validate`.

• **completion**: see in `common-attributes-completion`.

• **factor**: see in `common-attributes-factor`.

### image

Display an image.

• **name**: the name of the image. It must be the name of a record of `ir.ui.icon`.

• **yexpand**: see in `common-attributes-yexpand`.

• **yfill**: see in `common-attributes-yfill`.

• **colspan**: see in `common-attributes-colspan`.

• **states**: see in `common-attributes-states`.

• **help**: see in `common-attributes-help`.

### separator

Display a horizontal separator.

• **string**: The string that will be displayed above the separator.

• **name**: The name of the field from which the description will be used for string.

• **id**: see in `common-attributes-id`.

• **yexpand**: see in `common-attributes-yexpand`.

• **yfill**: see in `common-attributes-yfill`. 

---

6.1. Using tryond
- colspan: see in `common-attributes-colspan`.
- states: see in `common-attributes-states`.
- help: see in `common-attributes-help`.
- Requires that either `id` or `name` is defined.

**newline**

Force to use a new row.

**button**

Display a button.

- `string`: The string that will be displayed inside the button.
- `name`: The name of the function that will be called. The function must have this syntax:
  
  ```python
  button(cls, records)
  ```

  The function may return an *ir.action* id or one of those client side action keywords:
  
  - `new`: to create a new record
  - `delete`: to delete the selected records
  - `remove`: to remove the record if it has a parent
  - `copy`: to copy the selected records
  - `next`: to go to the next record
  - `previous`: to go to the previous record
  - `close`: to close the current tab
  - `switch <view type>`: to switch the view to the defined type
  - `reload`: to reload the current tab
  - `reload context`: to reload user context
  - `reload menu`: to reload menu
  - `icon`
  - `confirm`: A text that will be displayed in a confirmation popup when the button is clicked.
  - colspan: see in `common-attributes-colspan`.
  - states: see in `common-attributes-states`.
  - help: see in `common-attributes-help`.

**notebook**

It adds a notebook widget which can contain page tags.

- colspan: see in `common-attributes-colspan`.
- states: see in `common-attributes-states`.
Define a new tab inside a notebook.

- **string**: The string that will be displayed in the tab.
- **angle**: The angle in degrees between the baseline of the label and the horizontal, measured counterclockwise.
- **col**: see in `common-attributes-col`.
- **id**: see in `common-attributes-id`.
- **states**: see in `common-attributes-states`.
- Requires that either `id` or `name` is defined.

Create a sub-table in a cell.

- **string**: If set a frame will be drawn around the field with a label containing the string. Otherwise, the frame will be invisible.
- **rowspan**: The number of rows the group spans in the table.
- **col**: see in `common-attributes-col`.
- **homogeneous**: If True all the tables cells are the same size.
- **id**: see in `common-attributes-id`.
- **yexpand**: see in `common-attributes-yexpand`.
- **yfill**: see in `common-attributes-yfill`.
- **colspan**: see in `common-attributes-colspan`.
- **states**: see in `common-attributes-states`.
- Requires that either `id` or `name` is defined.

Contains the childs of a hpaned or vpaned.

**Example**
Tree view

The RNG that describes the xml for a tree view is stored in trytond/ir/ui/tree.rng. There is also a RNC in trytond/ir/ui/tree.rnc.

Tree view is used to display records inside a list or a tree.

It is a tree if there is a \texttt{field\_childs} defined and this tree will have drag and drop activated if the \texttt{field\_childs} and the parent field are defined in the view.

The columns of the view are put on the screen from left to right.

XML description

tree

Each tree view must start with this tag.

- \texttt{on\_write}: see \texttt{form\_attributes\_on\_write}.
- \texttt{editable}: If it is set to \texttt{top} or \texttt{bottom}, the list becomes editable and the new record will be added on top or bottom of the list.
- \texttt{sequence}: The name of the field that is used for sorting. This field must be an integer and it will be updated to match the new sort order when the user uses Drag and Drop on list rows.
- \texttt{keyword\_open}: A boolean to specify if the client should look for a tree_open action on double click instead of switching view.
• **tree_state**: A boolean to specify if the client should save the state of the tree.

**field**

• **name**: The name of the field.
• **readonly**: Boolean to set the field readonly.
• **widget**: The widget that must be used instead of the default one.
• **tree_invisible**: A string of *PYSON statement* that will be evaluated as boolean with the context of the view to display or not the column.
• **icon**: The name of the field that contains the name of the icon to display in the column.
• **sum**: A text for the sum widget that will be added on the bottom of list with the sum of all the fields in the column.
• **width**: Set the width of the column.
• **expand**: Boolean to specify if the column should be expanded to take available extra space in the view. This space is shared equally among all columns that have their “expand” property set to True. Resize don’t work if this option is enabled.
• **pre_validate**: see in *common-attributes-pre_validate*.
• **completion**: see in *common-attributes-completion*.
• **factor**: see in *common-attributes-factor*.

prefix or suffix

A *field* could contain one or many *prefix* or *suffix* that will be displayed in the same column.

• **string**: The text that will be displayed.
• **name**: The name of the field whose value will be displayed.
• **icon**: The name of the field that contains the name of the icon to display or the name of the icon.

**button**

Same as in *form-button*.

**Example**

```xml
<tree sequence="sequence">
    <field name="name"/>
    <field name="percentage">
        <suffix name="percentage" string="%"/>
    </field>
    <field name="group"/>
    <field name="type"/>
    <field name="active"/>
    <field name="sequence" tree_invisible="1"/>
</tree>
```
**button**

Display a button.

- **string**: The string that will be displayed inside the button.
- **name**: The name of the function that will be called. The function must have this syntax:
  
  ```python
  button(cls, records)
  ```
- **confirm**: A text that will be displayed in a confirmation popup when the button is clicked.
- **help**: see in `common-attributes-help`

**Graph view**

The RNG that describes the xml for a graph view is stored in `trytond/ir/ui/graph.rng`. There is also a RNC in `trytond/ir/ui/graph.rnc`.

**XML description**

**graph**

Each graph view must start with this tag.

- **type**: vbar, hbar, line, pie
- **background**: an hexadecimal value for the color of the background.
- **color**: the main color.
- **legend**: a boolean to specify if the legend must be displayed.

**x, y**

Describe the field that must be used for axis. **x** must contain only one tag **field** and **y** must at least one but may contain many.

**field**

- **name**: the name of the field on the object to use.
- **string**: allow to override the string that comes from the object.
- **key**: can be used to distinguish fields with the same name but with different domain.
- **domain**: a PySON string which is evaluated with the object value as context. If the result is true the field value is added to the graph.
- **fill**: defined if the graph shall be filled.
- **empty**: defined if the line graph must put a point for missing dates.
- **color**: the color of the field.
- **interpolation**: defined how the line graph must interpolate points. The default is `linear`.
  - **constant-center**: use the value of the nearest point, see Nearest-neighbor interpolation
– constant-left: use the value of the nearest left point.
– constant-right: use the value of the nearest right point.
– linear: see linear interpolation

Example

```xml
<graph string="Invoice by date" type="vbar">
  <x>
    <field name="invoice_date"/>
  </x>
  <y>
    <field name="total_amount"/>
  </y>
</graph>
```

Board view

The RNG that describes the xml for a board view is stored in tryond/ir/ui/board.rng. There is also a RNC in tryond/ir/ui/graph.rnc.

Board view is used to display multiple views at once.

Elements are put on the screen following the same rules as for Form view.

The views can be updated by the selection of records on another view inside the same board by using Eval() on the action id of the other view in the domain.

XML description

board

Each board view must start with this tag.

• col: see in common-attributes-col.

image

Same as in Form view.

separator

Same as in Form view.

label

Same as in Form view.
newline

Same as in Form view.

notebook

Same as in Form view.

page

Same as in Form view.

group

Same as in Form view.

hpaned, vpaned

Same as in Form view.

child

Same as in Form view.

action

- name: The id of the action window.
- colspan: see in common-attributes-colspan.

Calendar view

The RNG that describes the xml for a calendar view is stored in trytond/ir/ui/calendar.rng. There is also a RNC in trytond/ir/ui/calendar.rnc.

Calendar view is use to display records as events on a calendar based on a dtstart and optionally a dtend.

XML description

calendar

Each calendar view must start with this tag.
- dtstart: The name of the field that contains the start date.
- dtend: The name of the field that contains the end date.
• **mode**: An optional name for the view that will be used first. Available views are: *week* and *month*. The default value is *month*.

• **color**: An optional field name that contains the text color for the event. The default value is *black*.

• **background_color**: An optional field name that contains the background color for the event. The default value is *lightblue*.

### field

• **name**: The name of the field.

#### Example

```xml
<calendar dtstart="planned_date">
  <field name="code"/>
  <field name="product"/>
  <field name="reference"/>
</calendar>
```

### 6.1.15 Extending View

Extending a view means, that the original view will be modified by a set of rules which are defined with XML.

For this purpose, the extension engine uses xpath expressions.

The view is defined with the field *inherit* of the ir.ui.view.

If the field *domain* (a Python string representation of a *domain*) is not set or evaluated to True, the inheritance will be proceeded.

#### XML Description

**data**

Each view must start with this tag.

**xpath**

• **expr**: the xpath expression to find a node in the inherited view.

• **position**: Define the position in relation to the node found. It can be *before*, *after*, *replace*, *inside* or *replace_attributes* which will change the attributes.

#### Example

```xml
<data>
  <xpath
    expr="/form/notebook/page/separator[@name=&quot;signature&quot;]"
    position="before"/>
</data>
```
6.1.16 Wizard

A wizard describes a series of steps defined as `trytond.wizard.State`. The wizard stores data in `ir.session.wizard` between states.

The basics:

- Each wizard is a Python class that subclasses `trytond.wizard.Wizard`.
- The states of the wizard are attributes that are instances of `trytond.wizard.State`.

Example

This example defines a wizard which export translations

```python
from trytond.wizard import Wizard, StateView, StateTransition, Button
from trytond.pool import Pool

class TranslationExport(Wizard):
    __name__ = 'ir.translation.export'

    start = StateView('ir.translation.export.start',
                      'ir.translation_export_start_view_form', [
                        Button('Cancel', 'end', 'tryton-cancel'),
                        Button('Export', 'export', 'tryton-ok', default=True),
                      ])

    export = StateTransition()

    result = StateView('ir.translation.export.result',
                        'ir.translation_export_result_view_form', [
                        Button('Close', 'end', 'tryton-close'),
                      ])

    def transition_export(self):
        pool = Pool()
        translation_obj = pool.get('ir.translation')
        file_data = translation_obj.translation_export(
                        self.start.language.code, self.start.module.name)
        self.result.file = buffer(file_data)
        return 'result'

    def default_result(self, fields):
        return {
            'file': self.result.file,
        }
```

Pool.register(TranslationExport, type_='wizard')

The class must be registered in the Pool.

6.1.17 Reports

Tryton can generate dynamic reports in many formats from templates. The reports are generated in one step as follows: a report template in a special file format, explained later, is interpolated with dynamic data and placed into a document of the same file format. Tryton’s ability to generate documents in this way allows documents to be generated for any editor that supports the Open Document Format which can be converted to third party formats, such as PDF. Extra libraries are required for this, see INSTALL for more information.

Report Templates

Report templates are files with a format supported by relatorio, that contain snippets of the Genshi templating language. Here is an example of the text that would be placed in an open document text document, *.odt, that displays the full name and the address lines of the first address of each party. The genshi code is placed in the template using Functions->Placeholder->Text Fields. These are specific to ODT files.

Report API

Python API

XML Description

When defining an ir.action.report the following attributes are available:

- **name**: The name of the report.
- **report_name**: The name of the report model, for example my_module.my_report. This is the name you would use with Pool().get
- **model**: If this report is of an existing model this is its name. For example my_module.my_model. Custom reports that aren’t of a specific model will need to leave this blank.
- **report**: The path to the template file starting with the module, for example my_module/my_report.odt.
- **template_extension**: The template format.
- **single**: True if the template works only for one record. If such report is called with more than one record, a zip file containing all the reports will be generated.

Report Usage

Using genshi and open office reports

Setting up an ODT file

If you are creating a report from scratch you should perform the following steps:

- Remove user data
– “File > Properties…”
– Uncheck “Apply user data”
– Click on “Reset”

• Select Style and Formatting
  – Press F11 or “Format > Style and Formatting”
  – Click on the drop down at the right top
  – Select “Load Styles”
  – Click on “From File…”
  – Select a existing report (company/header_A4.odt)

• Set some parameters
  – Set the zoom to 100% (View>Zoom)
  – Set the document in read-only mode (File>Properties>Security) (Decreases the time it takes to open the document.)

• Usage
  – Use Liberation fonts (Only necessary if being officially included in Tryton)
  – Try to use styles in report templates so that they can be extended.

Using Genshi in an ODT file

The genshi code is placed in the template using Functions->Placeholder->Text Fields. These are specific to *.odt files and can be found in the open office menu at Insert -> Fields -> Other and then Functions -> Placeholder -> Text. Type genshi code into the Placeholder field. There are alternatives for embedding genshi that are supported by relatorio but their use is not encouraged within Tryton.

Also note that relatorio only supports a subset of genshi. The directives that are supported by relatorio can be found here: Quick Example.

See genshi’s documentation for more information: Genshi XML Templates

Examples

The modules company, account_invoice and stock all contain helpful examples.

Also see relatorio’s site for some examples:

• Quick Example
• In Depth Introduction
• Example Documents

Accessing models from within the report

By default instances of the models the report is for are passed in to the report via a list of objects called records (or record if single is True). These records behave just as they would within trytond itself. You can access any of the models relations as well. For example within the invoice report each object is an invoice and you can access the name
of the party of the invoice via `invoice.party.name`. Additional objects can be passed to a report. This is discussed below in *Passing custom data to a report*.

Within Tryton the underlying model the report can be found by following the Menu to Administration > UI > Actions > Report. Furthermore in tryton the fields for that model can be found by following the menu to Administration > Model > Model. Model relation fields can be accessed to any depth, for example, one could access `invoice.party.addresses` to get a list of addresses for the party of an invoice.

### Creating a simple report template for a model from client

TODO: Explain the steps necessary to do this.

### Creating a simple report template for a model in XML

Less work has to be done if you just want a simple report representation of a model. There are just 2 steps. First, create a report template file in a format supported by relatorio. Second, describe your report in XML making sure to define the correct `report_name` and `model`.

### Replacing existing Tryton reports

To replace an existing report you must deactivate the old report and activate the new report.

For example to deactivate the sale report:

```xml
<record model="ir.action.report" id="sale.report_sale">
    <field name="active" eval="False"/>
</record>
```

Then you must activate the new sale report that exists in your new module:

```xml
<record model="ir.action.report" id="report_sale">
    <field name="name">Sale</field>
    <field name="report_name">sale.sale</field>
    <field name="model">sale.sale</field>
    <field name="report">my_module/sale.odt</field>
    <field name="template_extension">odt</field>
</record>
```

And create the keyword for the new report:

```xml
<record model="ir.action.keyword" id="report_sale_keyword">
    <field name="keyword">form_print</field>
    <field name="model">sale.sale,-1</field>
    <field name="action" ref="report_sale"/>
</record>
```

### Passing custom data to a report

In this example `Report.get_context` is overridden and an employee object is set into context. Now the invoice report will be able to access the employee object.
from trytond.report import Report
from tryton.pool import Pool

class InvoiceReport(Report):
    __name__ = 'account.invoice'

    @classmethod
    def get_context(cls, records, data):
        pool = Pool()
        Employee = pool.get('company.employee')

        context = super(InvoiceReport, cls).get_context(records, data)
        employee_id = Transaction().context.get('employee')
        employee = Employee(employee_id) if employee_id else None
        context['employee'] = employee

        return context

Pool.register(InvoiceReport, type_='report')

6.1.18 Remote Procedure Call

There are two protocols supported by trytond: JSON-RPC (Version 1.0) and XML-RPC. The URL of the calls must end with the database name with a trailing ‘/’.

The available methods are:

**common.db.login**

It takes as parameters: the user name and a dictionary of login parameters. It returns in case of success the user ID and the session. If the parameters are not valid to authenticate the user, it returns nothing. Otherwise if it misses a key in the parameters, it raises a `LoginException` exception with the missing key name, type and the message to ask to the user.

**common.db.logout**

It takes no parameters and it invalidate the current session.

**Authorization**

Most of the calls require authorization, there are two methods:

**Basic**

It follows the Basic access authentication.

**Session**

The authorization field is constructed by the username, the user ID and the session combined with a single colon and encoded in Base64. The session is retrieved by calling the method `common.db.login`. 
6.1.19 User Application

Tryton provides a way to connect URL rules to a callable endpoint using the decorator method `route` of the `trytond.application.app` instance. This allows to define a custom API based on HTTP that can be used to create a specific user application.

The decorator takes as first parameter a string which follow the Rule Format of Werkzeug and as second parameter sequence of HTTP methods.

Example:

```python
from trytond.application import app
@app.route('/hello', methods=['GET'])
def hello(request):
    return 'Hello world'
```

Tryton also provides some wrappers in `trytond.protocols.wrappers` to ease the creation of such route.

- `set_max_request_size(size)`: change the default limit of the request to the size in bytes.
- `with_pool`: which takes the first parameter as database name and replace it by the corresponding instance of the `Pool`.
- `with_transaction([readonly])`: which starts a `Transaction` using the `Pool` from `with_pool`. If `readonly` is not set, the transaction will not be readonly for `POST`, `PUT`, `DELETE` and `PATCH` methods and readonly for all others.
- `user_application(name[, json])`: which set the `Transaction.user` from the `Authorization` header using the type `bearer` and a valid key for the named user application.

User Application Key

Tryton also provides a easy way to manage access to user application using keys per named application. A key is created with a `POST` request on the URL `/<database_name>/user/application/` which returns the key. The request must contain as data a json object with the keys:

- `user`: the user login
- `application`: the name of the application

After the creation, the key must be validated by the user from the preferences of a Tryton client.

A key can be deleted with a `DELETE` request on the same `URL`. The request must contain as data a json object with the keys:

- `user`: the user login
- `key`: the key to delete
- `application`: the name of the application of the key

6.1.20 Modules

The modules of Tryton extend the functionality of the platform. The server comes by default with only a basic functionality included in these modules: `ir, res`. 
Module Structure

A module is a directory in tryond/modules which contains at least two files:

- __init__.py: a Tryton module must be a Python module.
- tryton.cfg: a Configuration file that describes the Tryton module.

__init__.py file

It is the Python __init__.py to define a module. It must contains a method named register() that must register to the pool all the objects of the module.

tryton.cfg file

It is a configuration file using the format of ConfigParser that must contain tryton section with this following name:

- version: The version number of the module.
- depends: A one per line list of modules on which this module depends.
- extras_depend: A one per line list of modules on which this module may depend.
- xml: The one per line list of the XML files of the module. They will be loaded in the given order at the installation or update of the module.

Here is an example:

```plaintext
[tryton]
version=0.0.1
depends:
  ir
  res
country
xml:
  party.xml
category.xml
address.xml
  contact_mechanism.xml
```

Python Files

The Python files define the models for the modules.

XML Files

The XML files define data that will be inserted into the database.

There is an rnc for those files stored in tryond/tryton.rnc.

The following snippet gives a first idea of what an xml file looks:
Here is the list of the tags:

- **tryton**: The main tag of the xml
- **data**: Define a set of data inside the file. It can have the attributes:
  - `noupdate` to prevent the framework to update the records,
  - `skiptest` to prevent import of data when running tests,
  - `depends` to import data only if all modules in the comma separated module list value are installed,
  - `grouped` to create records at the end with a grouped call.
- **record**: Create a record of the model defined by the attribute `model` in the database. The `id` attribute can be used to refer to the record later in any xml file.
- **field**: Set the value of the field with the name defined by the attribute `name`.

Here is the list of attributes:

- **search**: Only for relation field. It contains a domain which is used to search for the value to use. The first value found will be used.
- **ref**: Only for relation field. It contains an xml id of the relation to use as value. It must be prefixed by the module name with an ending dot, if the record is defined in another module.
– eval: Python code to evaluate and use result as value. The following expressions are available:
  * time: The python time module
  * version: The current Tryton version
  * ref: A function that converts an XML id into a database id.
  * Decimal: The python Decimal object
  * datetime: The python datetime module
– pyson: convert the evaluated value into PYSON string.

Note: Field content is considered as a string. So for fields that require other types, it is required to use the eval attribute.

• menuitem: Shortcut to create ir.ui.menu records.

Here is the list of attributes:
  – id: The id of the menu.
  – name: The name of the menu.
  – icon: The icon of the menu.
  – sequence: The sequence value used to order the menu entries.
  – parent: The xml id of the parent menu.
  – action: The xml id of the action linked to the menu.
  – groups: A list of xml id of group, that have access to the menu, separated by commas.
  – active: A boolean telling if the menu is active or not.

6.1.21 Translation

The translation of the user interface is provided module-wise. Translations are stored in the locale/ directory of a module, each language in a PO-file. The official language files are named after the POSIX locale standard, e.g. de_DE.po, es.po, es_AR.po, es_EC.po... The names of custom language files must match the code of the language in the Model ir.lang.

If a language is set translatable, the translations will be loaded into the database on each trytond module update. Tryton supports derivative translations. This means that if the translation of a term is missing in one language, it will search on the parent languages. Also when activate a children language, you must also activate all parents.

Translation Wizards

Set Report Translations Wizard

The wizard adds new translations to the base language en.

Clean Translations Wizard

The wizard deletes obsolete translations from the database.
Synchronize Translations Wizard

The wizard updates the translations of the selected language based on the translations of the base language en. It will also remove duplicate translations with its direct parent.

Export Translations Wizard

The wizard requires to select a language and a module and will export the translations for this selection into a PO-file.

Override translations

Translations of a module can be overridden by another module. This can be done by putting a PO file into the locale/override directory of the module that shall contain the translations to override.

To override the translation of another module the msgctxt string must have the following content:

```
type:name:module.xml_id
```

- **type**: Value of the field type of ir.translation.
- **name**: Value of the field name of ir.translation.
- **module**: Value of the field module ir.translation.
- **xml_id**: The XML id that is stored in ir.model.data as fs_id

The xml_id part is optional and can be omitted if it is None.

6.1.22 Testing

Tryton supports both functional and unit tests.

Testing your module

Functional tests

Functional tests are written as doctests using proteus.

Unit tests

Tryton provides the ModuleTestCase class that bundles a set of tests that are useful for every module.

Unit tests in ModuleTestCase can be decorated with trytond.tests.test_tryton.with_transaction() to run the test in a transaction.

To use it in your own module you just have to inherit from ModuleTestCase and set the class attribute module to the name of your module.

```python
from trytond.tests.test_tryton import ModuleTestCase, with_transaction

class MyModuleTestCase(ModuleTestCase):
    "My Module Test Case"
    module = 'my_module'
```

(continues on next page)
def test_method(self):
    "Test method"
    self.assert_(True)

Tests from this modules are found by the function trytond.modules.my_module.tests.suite which must return a unittest.TestSuite containing all the module’s tests. This function is called by the Tryton test runner script to gather all the tests.

A typical suite() function thus looks like this:

def suite():
    suite = trytond.tests.test_tryton.suite()
    suite.addTests(unittest.TestLoader().loadTestsFromTestCase(
        MyModuleTestCase))
    suite.addTests(doctest.DocFileSuite('scenario_my_module.rst',
        tearDown=doctest_teardown, encoding='utf-8',
        optionflags=doctest.REPORT_ONLY_FIRST_FAILURE,
        checker=doctest_checker))
    return suite

Running your module’s tests

Tryton provides the script trytond/tests/run-tests.py, just invoke it like that:

run-tests.py -m my_module

Testing trytond

Extending tests

Python modules extending tryton core can define additional tests that should be added to the existing ones. Those modules must create an entry point trytond.tests. Any file in the module path specified by this entry point starting with test_ and ending by .py will be imported. Each of those file must define a suite() function that returns a unittest.TestSuite that will be included in the trytond test suite. If the module from the entry point defines a register function it will be called when registering the test-specific models in the tryond.pool.Pool.

Running trytond tests

You should use the script trytond/tests/run-tests.py by invoking it like that:

run-tests.py [-c configuration]

You can use a different configuration file to check trytond against different backend.
6.2 API Reference

6.2.1 Models

Model API reference.

**Model**

```python
class trytond.model.Model([id[,**kwargs]])
```

This is the base class that every kind of `model` inherits. It defines common attributes of all models.

Class attributes are:

**Model.**`__name__`

It contains a unique name to reference the model throughout the platform.

**Model.**`__rpc__`

It contains a dictionary with method name as key and an instance of `trytond.rpc.RPC` as value.

**Model.**`_error_messages`

It contains a dictionary mapping keywords to an error message. By way of example:

```python
_error_messages = {
    'recursive_categories': 'You can not create recursive categories!',
    'wrong_name': 'You can not use " / " in name field!'
}
```

**Model.**`_rec_name`

It contains the name of the field used as name of records. The default value is `name`.

**Model.**`id`

The definition of the field `id` of records.

Class methods:

**classmethod Model.**`__setup__()`

Setup the class before adding into the `trytond.pool.Pool`.

**classmethod Model.**`__post_setup__()`

Setup the class after added into the `trytond.pool.Pool`.

**classmethod Model.**`__register__(module_name)`

Registers the model in `ir.model` and `ir.model.field`.

**classmethod Model.**`raise_user_error(error[, error_args[, error_description[, error_description_args[, raise_exception]]]]])`

Raises an exception that will be displayed as an error message in the client. `error` is the key of the error message in `error_messages` and `error_args` is the arguments for the “%”-based substitution of the error message. There is the same parameter for an additional description. The boolean `raise_exception` can be set to `False` to retrieve the error message strings.

**classmethod Model.**`raise_user_warning(warning_name[, warning[, warning_args[, warning_description[, warning_description_args]]]]])`

Raises an exception that will be displayed as a warning message on the client, if the user has not yet bypassed it. `warning_name` is used to uniquely identify the warning. Others parameters are like in `Model.raise_user_error()`.
Warning: It requires that the cursor will be commited as it stores state of the warning states by users.

classmethod Model.default_get(fields_names[, with_rec_name])
Returns a dictionary with the default values for each field in fields_names. Default values are defined by the returned value of each instance method with the pattern default_`field_name`() with_rec_name allow to add rec_name value for each many2one field. The default_rec_name key in the context can be used to define the value of the Model._rec_name field.

classmethod Model.fields_get([fields_names])
Return the definition of each field on the model.

Instance methods:

Model.on_change(fieldnames)
Returns the list of changes by calling on_change method of each field.

Model.on_change_with(fieldnames)
Returns the new values of all fields by calling on_change_with method of each field.

Model.pre_validate()
This method is called by the client to validate the instance.

ModelView
class trytond.model.ModelView
It adds requirements to display a view of the model in the client.

Class attributes:

ModelView._buttons
It contains a dictionary with button name as key and the states dictionary for the button. This states dictionary will be used to generate the views containing the button.

Static methods:

static ModelView.button()
Decorate button method to check group access and rule.

static ModelView.button_action(action)
Same as ModelView.button() but return the action id of the XML id action.

static ModelView.button_change([fields])
Same as ModelView.button() but for button that change values of the fields on client side (similar to on_change).

Warning: Only on instance methods.

Class methods:

classmethod ModelView.fields_view_get([view_id[, view_type[, toolbar]]])
Return a view definition used by the client. The definition is:

```python
{
    'model': model name,
    'type': view type,
    'view_id': view id,
}
```
(continues on next page)
'arch': XML description,
'fields': {
  field name: {
  ...
  },

'field_childs': field for tree,
}

classmethod ModelView.view_toolbar_get()

Returns the model specific actions in a dictionary with keys:

• print: a list of available reports
• action: a list of available actions
• relate: a list of available relations

classmethod ModelView.view_attributes()

Returns a list of XPath, attribute and value. Each element from the XPath will get the attribute set with the JSON encoded value.

ModelStorage

class trytond.model.ModelStorage

It adds storage capability.

Class attributes are:

ModelStorage.create_uid

The definition of the trytond.model.fields.Many2One field create_uid of records. It contains the id of the user who creates the record.

ModelStorage.create_date

The definition of the trytond.model.fields.DateTime field create_date of records. It contains the datetime of the creation of the record.

ModelStorage.write_uid

The definition of the trytond.model.fields.Many2One field write_uid of the records. It contains the id of the last user who writes on the record.

ModelStorage.write_date

The definition of the trytond.model.fields.DateTime field write_date of the records. It contains the datetime of the last write on the record.

ModelStorage.rec_name

The definition of the trytond.model.fields.Function field rec_name. It is used in the client to display the records with a single string.

ModelStorage._constraints


The list of constraints that each record must respect. The definition is:
['function name', 'error keyword'], ... ]

where function name is the name of an instance or a class method of the which must return a boolean (False when the constraint is violated) and error keyword is a key of Model._error_messages.

Static methods:

```python
static ModelStorage.default_create_uid()
Return the default value for create_uid.
```

```python
static ModelStorage.default_create_date()
Return the default value for create_date.
```

Class methods:

```python
classmethod ModelStorage.create(vlist)
Create records. vlist is list of dictionaries with fields names as key and created values as value and return the list of new instances.
```

```python
classmethod ModelStorage.trigger_create(records)
Trigger create actions. It will call actions defined in ir.trigger if on_create is set and condition is true.
```

```python
classmethod ModelStorage.read(ids [fields_names])
Return a list of values for the ids. If fields_names is set, there will be only values for these fields otherwise it will be for all fields.
```

```python
classmethod ModelStorage.write(records, values [[records, values], ...])
Write values on the list of records. values is a dictionary with fields names as key and writen values as value.
```

```python
classmethod ModelStorage.trigger_write_get_eligibles(records)
Return eligible records for write actions by triggers. This dictionary is to pass to trigger_write().
```

```python
classmethod ModelStorage.trigger_write(eligibles)
Trigger write actions. It will call actions defined in ir.trigger if on_write is set and condition was false before write() and true after.
```

```python
classmethod ModelStorage.delete(records)
Delete records.
```

```python
classmethod ModelStorage.trigger_delete(records)
Trigger delete actions. It will call actions defined in ir.trigger if on_delete is set and condition is true.
```

```python
classmethod ModelStorage.copy(records [default])
Duplicate the records. default is a dictionary of default value for the created records. New records are returned following the input order.
```

```python
classmethod ModelStorage.search(domain [offset, limit, order, count])
Return a list of records that match the domain.
```

```python
classmethod ModelStorage.search_count(domain)
Return the number of records that match the domain.
```

```python
classmethod ModelStorage.search_read(domain [offset, limit, order, fields_names])
Call search() and read() at once. Useful for the client to reduce the number of calls.
```

```python
classmethod ModelStorage.search_rec_name(name, clause)
Searcher for the trytond.model.fields.Function field rec_name.
```

```python
classmethod ModelStorage.search_global(cls, text)
Yield tuples (record, name, icon) for records matching text. It is used for the global search.
```
classmethod ModelStorage.browse(ids)
    Return a list of record instance for the ids.

classmethod ModelStorage.export_data(records, fields_names)
    Return a list of list of values for each records. The list of values follows fields_names. Relational fields
    are defined with / at any depth. Descriptor on fields are available by appending . and the name of the method
    on the field that returns the descriptor.

classmethod ModelStorage.import_data(fields_names, data)
    Create records for all values in datas. The field names of values must be defined in fields_names. It
    returns the number of imported records.

classmethod ModelStorage.check_xml_record(records, values)
    Verify if the records are originating from XML data. It is used to prevent modification of data coming from
    XML files. This method must be overiden to change this behavior.

classmethod ModelStorage.check_recursion(records, parent)
    Helper method that checks if there is no recursion in the tree composed with parent as parent field name.

classmethod ModelStorage.validate(records)
    Validate the integrity of records after creation and modification. This method must be overridden to add valida-
    tion and must raise an exception if validation fails.

Dual methods:

classmethod ModelStorage.save(records)
    Save the modification made on the records.

Instance methods:

ModelStorage.get_rec_name(name)
    Getter for the trytond.model.fields.Function field rec_name.

ModelSQL

class trytond.model.ModelSQL
    It implements ModelStorage for an SQL database.

Class attributes are:

ModelSQL._table
    The name of the database table which is mapped to the class. If not set, the value of Model._name is used
    with dots converted to underscores.

ModelSQL._order
    The default order parameter of ModelStorage.search() method.

ModelSQL._order_name
    The name of the field (or an SQL statement) on which the records must be sorted when sorting on this model
    from an other model. If not set, ModelStorage._rec_name will be used.

ModelSQL._history
    If true, all changes on records will be stored in a history table.

ModelSQL._sql_constraints
    A list of SQL constraints that are added on the table:
        [ ('constraint name', constraint, 'error message key'), ... ]

    • constraint name is the name of the SQL constraint in the database
• constraint is an instance of Constraint
• error message key is the key of _sql_error_messages

ModelSQL._sql_error_messages
Like Model._error_messages but for _sql_constraints

Class methods:

classmethod ModelSQL.__table__()  
Return a SQL Table instance for the Model.

classmethod ModelSQL.__table_history__()  
Return a SQL Table instance for the history of Model.

classmethod ModelSQL.table_query()  
Could be overridden to use a custom SQL query instead of a table of the database. It should return a SQL FromItem.

**Warning:** By default all CRUD operation will raise an error on models implementing this method so the create, write and delete methods may also be overridden if needed.

classmethod ModelSQL.history_revisions(ids)  
Return a sorted list of all revisions for ids. The list is composed of the date, id and username of the revision.

classmethod ModelSQL.restore_history(ids, datetime)  
Restore the record ids from history at the specified date time. Restoring a record will still generate an entry in the history table.

**Warning:** No access rights are verified and the records are not validated.

classmethod ModelSQL.restore_history_before(ids, datetime)  
Restore the record ids from history before the specified date time. Restoring a record will still generate an entry in the history table.

**Warning:** No access rights are verified and the records are not validated.

classmethod ModelSQL.search(domain[, offset[, limit[, order[, count[, query]]]]]])  
Return a list of records that match the domain.

If offset or limit are set, the result starts at the offset and has the length of the limit.

The order is a list of tuples defining the order of the result:

```python
['field name', 'ASC'], ('other field name', 'DESC'), ... ]
```

The first element of the tuple is a field name of the model and the second is the sort ordering as ASC for ascending or DESC for descending. This second element may contain ‘NULLS FIRST’ or ‘NULLS LAST’ to sort null values before or after non-null values. If neither is specified the default behavior of the backend is used.

In case the field used is a fields.Many2One, it is also possible to use the dotted notation to sort on a specific field from the target record.

If count is set to True, then the result is the number of records.

If query is set to True, the the result is the SQL query.
classmethod ModelSQL.search_domain(domain[, active_test[, tables ]])
Convert a domain into a SQL expression by returning the updated tables dictionary and a SQL expression.

Where tables is a nested dictionary containing the existing joins:

```python
{
    None: (<Table invoice>, None),
    'party': {
        None: (<Table party>, <join_on sql expression>),
        'addresses': {
            None: (<Table address>, <join_on sql expression>),
            },
    },
}
```

Constraint

```python
class trytond.model.Constraint(table)
```
It represents a SQL constraint on a table of the database and it follows the API of the python-sql expression.

Instance attributes:

Constraint.table
The SQL Table on which the constraint is defined.

Check

```python
class trytond.model.Check(table, expression)
```
It represents a check Constraint which enforce the validity of the expression.

Instance attributes:

Check.expression
The SQL expression to check.

Unique

```python
class trytond.model.Unique(table, *columns)
```
It represents a unique Constraint which enforce the uniqueness of the group of columns with respect to all the rows in the table.

Instance attributes:

Unique.columns
The tuple of SQL Column instances.

Workflow

```python
class trytond.model.Workflow
```
A Mix-in class to handle transition check.

Class attribute:
Workflow._transition_state
   The name of the field that will be used to check state transition.

Workflow._transitions
   A set containing tuples of from and to state.

Static methods:

\[\text{static Workflow.transition(state)}\]
   Decorate method to filter ids for which the transition is valid and finally to update the state of the filtered ids.

ModelSingleton

class trytond.model.ModelSingleton
Modify ModelStorage into a singleton. This means that there will be only one record of this model. It is commonly used to store configuration value.

Class methods:

\[\text{classmethod ModelSingleton.get_singleton()}\]
   Return the instance of the unique record if there is one.

DictSchemaMixin

class trytond.model.DictSchemaMixin
A mixin for the schema of trytond.model.fields.Dict field.

Class attributes are:

DictSchemaMixin.name
   The definition of the trytond.model.fields.Char field for the name of the key.

DictSchemaMixin.string
   The definition of the trytond.model.fields.Char field for the string of the key.

DictSchemaMixin.type_
   The definition of the trytond.model.fields.Selection field for the type of the key. The available types are:
   • boolean
   • integer
   • char
   • float
   • numeric
   • date
   • datetime
   • selection

DictSchemaMixin.digits
   The definition of the trytond.model.fields.Integer field for the digits number when the type is float or numeric.
DictSchemaMixin.selection

The definition of the `trytond.model.fields.Text` field to store the couple of key and label when the type is `selection`. The format is a key/label separated by “:” per line.

DictSchemaMixin.selection_sorted

If the `selection` must be sorted on label by the client.

DictSchemaMixin.selection_json

The definition of the `trytond.model.fields.Function` field to return the JSON version of the `selection`.

Static methods:

```python
static DictSchemaMixin.default_digits()  # Return the default value for digits.
```

Class methods:

```python
classmethod DictSchemaMixin.get_keys(records)  # Return the definition of the keys for the records.
```

Instance methods:

```python
DictSchemaMixin.get_selection_json(name)  # Getter for the `selection_json`.
```

**MatchMixin**

```python
class trytond.model.MatchMixin
```

A mixin to add to a `Model` a match method on pattern. The pattern is a dictionary with field name as key and the value to compare. The record matches the pattern if for all dictionary entries, the value of the record is equal or not defined.

Instance methods:

```python
MatchMixin.match(pattern[, match_none])  # Return if the instance match the pattern. If match_none is set None value of the instance will be compared.
```

**UnionMixin**

```python
class trytond.model.UnionMixin
```

A mixin to create a `ModelSQL` which is the UNION of some `ModelSQL`'s. The ids of each models are sharded to be unique.

Static methods:

```python
static UnionMixin.union_models()  # Return the list of `ModelSQL`'s names
```

Class methods:

```python
classmethod UnionMixin.union_shard(column, model)  # Return a SQL expression that shards the column containing record id of model name.
classmethod UnionMixin.union_unshard(record_id)  # Return the original instance of the record for the sharded id.
classmethod UnionMixin.union_column(name, field, table, Model)  # Return the SQL column that corresponds to the field on the union model.
```
trytond Documentation, Release 4.6

classmethod UnionMixin.union_columns(model)
    Return the SQL table and columns to use for the UNION for the model name.

sequence_ordered

trytond.model.sequence_ordered([field_name, field_label, order])
    Returns a mixin class which defines the order of a ModelSQL with an trytond.model.fields.Integer field. field_name indicates the name of the field to be created and its default values is sequence. field_label defines the label which will be used by the field and defaults to Sequence. Order specifies the order direction and defaults to ASC NULLS FIRST.

MultiValueMixin

class trytond.model.MultiValueMixin
    A mixin for Model to help having trytond.model.fields.MultiValue fields with multi-values on a ValueMixin. The values are stored by creating one record per pattern. The patterns are the same as those on MatchMixin.

    Class methods:
    classmethod MultiValueMixin.multivalue_model(field)
        Return the ValueMixin on which the values are stored for the field name. The default is class name suffixed by the field name.
    classmethod MultiValueMixin.setter_multivalue(records, name, value, **pattern)
        The setter method for the trytond.model.fields.Function fields.

    Instance methods:
    MultiValueMixin.multivalue_records(field)
        Return the list of all ValueMixin records linked to the instance. By default, it returns the value of the first found trytond.model.fields.One2Many linked to the multivalue model or all the records of this one.
    MultiValueMixin.multivalue_record(field, **pattern)
        Return a new record of ValueMixin linked to the instance.
    MultiValueMixin.get_multivalue(name, **pattern)
        Return the value of the field name for the pattern.
    MultiValueMixin.set_multivalue(name, value, save, **pattern)
        Store the value of the field name for the pattern. If save is true, it will be stored in the database, otherwise the modified ValueMixin records are returned unsaved. save is true by default.

    Warning: To customize the pattern, both methods must be override the same way.

ValueMixin

class trytond.model.ValueMixin
    A mixin to store the values of MultiValueMixin.
Fields

Fields define the behavior of the data on model’s record.

Field options

The following arguments are available to all field types. All are optional except Field.string.

string

Field.string
A string for the label of the field.

help

Field.help
A multi-line help string for the field.

required

Field.required
If True, the field is not allowed to be empty. Default is False.

readonly

Field.readonly
If True, the field is not editable in the client. Default is False.

Warning: For relational fields, it means only the new, delete, add and remove buttons are inactivated. The editable state of the target record must be managed at the target model level.

domain

Field.domain
A domain constraint that will be applied on the field value.

states

Field.states
A dictionary that defines dynamic states of the field and overrides the static one. Possible keys are required, readonly and invisible. The values are PYSON statements that will be evaluated with the values of the record.
select

Field.select
If true, the content of the field will be indexed.

on_change

Field.on_change
A set of field names. If this attribute is set, the client will call the method on_change_<field name> of the model when the user changes the current field value and will give the values of each fields in this list. The method signature is:

```python
on_change_<field name>()
```

This method must change the value of the fields to be updated.

Note: The on_change_<field name> methods are running in a rollbacked transaction.

The set of field names could be filled by using the decorator depends().

on_change_with

Field.on_change_with
A set of field names. Same like on_change, but defined the other way around. If this attribute is set, the client will call the method on_change_with_<field name> of the model when the user changes one of the fields defined in the list and will give the values of each fields in this list. The method signature is:

```python
on_change_with_<field name>()
```

This method must return the new value of the field.

Note: The on_change_with_<field name> methods are running in a rollbacked transaction.

The set of field names could be filled by using the decorator depends().

depends

Field.depends
A list of field names on which the current one depends. This means that the client will also read these fields even if they are not defined on the view. Field.depends is used per example to ensure that PYSON statement could be evaluated.

context

Field.context
A dictionary which will update the current context for *relation field*.

**Warning:** The context could only depend on direct field of the record and without context.

**loading**

Field.**loading**

Define how the field must be loaded: lazy or eager.

**name**

Field.**name**

The name of the field.

Instance methods:

Field.**convert_domain**(domain, tables, Model)

Convert the simple *domain* clause into a SQL expression or a new domain. *tables* could be updated to add new joins.

Field.**sql_format**(value)

Convert the value to use as parameter of SQL queries.

Field.**sql_type**()

Return the nameduple(`SQLType`, ‘base type’) which defines the SQL type to use for creation and casting. Or *None* if the field is not stored in the database.

- sql_type is using the `_sql_type` attribute to compute its return value. The backend is responsible for the computation.

- For the list of supported types by Tryton see backend types.

Field.**sql_column**(table)

Return the Column instance based on table.

Field.**set_rpc**(model)

Adds to *model* the default RPC instances required by the field.

**Default value**

See *default value*

**Searching**

A class method could be defined for each field which must return a SQL expression for the given domain instead of the default one. The method signature is:

```
domain_<field name>(domain, tables)
```

Where *domain* is the simple *domain* clause and *tables* is a nested dictionary, see `convert_domain()`.
Ordering

A class method could be defined for each field which must return a list of SQL expression on which to order instead of the field. The method signature is:

```
order_<field name>(tables)
```

Where `tables` is a nested dictionary, see `convert_domain()`.

Depends

```
trytond.model.fields.depends([*fields[, methods ]])
```

A decorator to define the field names on which the decorated method depends. The `methods` argument can be used to duplicate the field names from other fields. This is useful if the decorated method calls another method.

Field types

Boolean

```
class trytond.model.fields.Boolean(string[, **options ])
```

A true/false field.

Integer

```
class trytond.model.fields.Integer(string[, **options ])
```

An integer field.

BigInteger

```
class trytond.model.fields.BigInteger(string[, **options ])
```

A long integer field.

Char

```
class trytond.model.fields.Char(string[, size[, translate[, **options ]]])
```

A single line string field. `Char` has two extra optional arguments:

- **Char.size**
  The maximum length (in characters) of the field. The size is enforced at the storage level and in the client input.

- **Char.translate**
  If true, the value of the field is translatable. The value read and stored will depend on the `language` defined in the context.
Char.autocomplete
A set of field names. If this attribute is set, the client will call the method autocomplete_<field name> of the model when the user changes one of those field value. The method signature is:

```python
autocomplete_<field name>()
```

This method must return a list of string that will populate the ComboboxEntry in the client. The set of field names could be filled by using the decorator `depends()`.

**Text**

```python
class trytond.model.fields.Text(string[, size[, translatable[, **options ]]]])
```

A multi line string field.

*Text* has two extra optional arguments:

- **Text.size**
  - Same as `Char.size`

- **Text.translate**
  - Same as `Char.translate`

**Float**

```python
class trytond.model.fields.Float(string[, digits[, **options ]])
```

A floating-point number field. It will be represented in Python by a `float` instance.

*Float* has one extra optional arguments:

- **Float.digits**
  - A tuple of two integers. The first integer defines the total of numbers in the integer part. The second integer defines the total of numbers in the decimal part. Integers can be replaced by a `PYSON` statement. If digits is None or any values of the tuple is None, no validation on the numbers will be done.

**Numeric**

```python
class trytond.model.fields.Numeric(string[, digits[, **options ]])
```

A fixed-point number field. It will be represented in Python by a `decimal.Decimal` instance.

*Numeric* has one extra optional arguments:

- **Numeric.digits**
  - Same as `Float.digits`

**Date**

```python
class trytond.model.fields.Date(string[, **options ])
```

A date, represented in Python by a `datetime.date` instance.
**DateTime**

```python
class trytond.model.fields.DateTime(string[, format, **options])
```

A date and time, represented in Python by a `datetime.datetime` instance. It is stored in UTC while displayed in the user timezone.

*DateTime.format*

A string format as used by `strftime`. This format will be used to display the time part of the field. The default value is `%H:%M:%S`. The value can be replaced by a `PYSON` statement.

**Timestamp**

```python
class trytond.model.fields.Timestamp(string[, **options])
```

A timestamp, represented in Python by a `datetime.datetime` instance.

**Time**

```python
class trytond.model.fields.Time(string[, format, **options])
```

A time, represented in Python by a `datetime.time` instance.

*Time.format*

Same as *DateTime.format*

**TimeDelta**

```python
class trytond.model.fields.TimeDelta(string[, converter[, **options]])
```

An interval, represented in Python by a `datetime.timedelta` instance.

*TimeDelta.converter*

The name of the context key containing the time converter. A time converter is a dictionary with the keys: `s` (second), `m` (minute), `h` (hour), `d` (day), `w` (week), `M` (month), `Y` (year) and the value in second.

**Binary**

```python
class trytond.model.fields.Binary(string[, **options])
```

A binary field. It will be represented in Python by a `bytes` instance.

---

**Warning:** If the context contains a key composed of the model name and field name separated by a dot and its value is the string `size` then the read value is the size instead of the content.

*Binary* has three extra optional arguments:

*Binary.filename*

Name of the field that holds the data’s filename. Default value is an empty string, which means the data has no filename (in this case, the filename is hidden, and the “Open” button is hidden when the widget is set to “image”).
Binary **file_id**
Name of the field that holds the FileStore identifier. Default value is *None* which means the data is stored in the database. The field must be on the same table and accept *char* values.

**Warning:** Switching from database to file-store is supported transparently. But switching from file-store to database is not supported without manually upload to the database all the files.

Binary **store_prefix**
The prefix to use with the FileStore. Default value is *None* which means the database name is used.

### Selection

class trytond.model.fields.Selection

A string field with limited values to choice.

**Selection** has one extra required argument:

**Selection**. **selection**
A list of 2-tuples that looks like this:

```python
['M', 'Male'],
['F', 'Female'],
```

The first element in each tuple is the actual value stored. The second element is the human-readable name.

It can also be the name of a class or instance method on the model, that will return an appropriate list. The signature of the method is:

```python
selection()
```

**Note:** The method is automatically added to trytond.model.Model._rpc if not manually set.

**Selection** has two extra optional arguments:

**Selection**. **sort**
If true, the choices will be sorted by human-readable value. Default value is *True*.

**Selection**. **selection_change_with**
A set of field names. If this attribute is set, the client will call the selection method of the model when the user changes on of the fields defined in the list and will give the values of each fields in the list. The selection method should be an instance method. The set of field names could be filled by using the decorator *depends()*.

**Selection**. **translate_selection**
If true, the human-readable values will be translated. Default value is *True*.

Instance methods:

**Selection**. **translated**
Returns a descriptor for the translated value of the field. The descriptor must be used on the same class as the field. It will use the language defined in the context of the instance accessed.
Reference

class trytond.model.fields.Reference(string[, selection[, selection_change_with[, **options]]])

A field that refers to a record of a model. It will be represented in Python by a str instance like this:

'\<model name>,<record id>'

But a tuple can be used to search or set value.

Reference has three extra optional arguments:

Reference.selection
   Same as Selection.selection but only for model name.

Reference.selection_change_with
   Same as Selection.selection_change_with.

Reference.datetime_field
   Same as Many2One.datetime_field

Instance methods:

Reference.translated([name])
   Same as translated() but for the translated name of the target model.

Many2One

class trytond.model.fields.Many2One(model_name, string[, left[, right[, ondelete[, datetime_field[, target_search[, **options]]]]]])

A many-to-one relation field.

Many2One has one extra required argument:

Many2One.model_name
   The name of the target model.

Many2One has some extra optional arguments:

Many2One.left
   The name of the field that stores the left value for the Modified Preorder Tree Traversal. It only works if the model_name is the same then the model.

   Warning: The MPTT Tree will be rebuild on database update if one record is found having left or right field value equals to the default or NULL.

Many2One.right
   The name of the field that stores the right value. See left.

Many2One.ondelete
   Define the behavior of the record when the target record is deleted. Allowed values are:
   • CASCADE: it will try to delete the record.
   • RESTRICT: it will prevent the deletion of the target record.
   • SET NULL: it will empty the relation field.
SET NULL is the default setting.

Note: SET NULL will be override into RESTRICT if required is true.

Many2One.datetime_field
If set, the target record will be read at the date defined by the datetime field name of the record. It is usually used in combination with trytond.model.ModelSQL._history to request a value for a given date and time on a historicized model.

Many2One.target_search
Define the kind of SQL query to use when searching on related target. Allowed values are:
  • subquery: it will use a subquery based on the ids.
  • join: it will add a join on the main query.
join is the default value.

Note: join could improve the performance if the target has a huge amount of records.

One2Many

class trytond.model.fields.One2Many (model_name, field, string[, add_remove[, order[, datetime_field[, size[, **options]]]]]])
A one-to-many relation field. It requires to have the opposite Many2One field or a Reference field defined on the target model.

This field accepts as written value a list of tuples like this:
  • (create[, [{<field name>: value, ...}, ...]]): it will create new target records and link them to this one.
  • (write[[], ids,...], {<field name>: value, ...}, ...]): it will write values to target ids.
  • (delete[[], ids,...]): it will delete the target ids.
  • (add[[], ids,...]): it will link the target ids to this record.
  • (remove[[], ids,...]): it will unlink the target ids from this record.
  • (copy, ids[, {<field name>: value, ...}]): it will copy the target ids to this record. Optional field names and values may be added to override some of the fields of the copied records.

One2Many has some extra required arguments:

One2Many.model_name
The name of the target model.

One2Many.field
The name of the field that handles the opposite Many2One or Reference.

One2Many has some extra optional arguments:

One2Many.add_remove
A domain to select records to add. If set, the client will allow to add/remove existing records instead of only create/delete.
One2Many.

**filter**

A *domain* that is not a constraint but only a filter on the records.

One2Many.

**order**

A list of tuple defining the default order of the records like for `trytond.model.ModelSQL._order`.

One2Many.

**datetime_field**

Same as `Many2One.datetime_field`.

One2Many.

**size**

An integer or a PYSON expression denoting the maximum number of records allowed in the relation.

---

Many2Many

```python
class trytond.model.fields.Many2Many(relation_name, origin, target, string[, order[, datetime_field[, size[, **options]]]])
```

A many-to-many relation field. It requires to have the opposite origin `Many2One` field or a `:class:`Reference field defined on the relation model and a `Many2One` field pointing to the target.

This field accepts as written value a list of tuples like the `One2Many`.

`Many2Many` has some extra required arguments:

Many2Many.

**relation_name**

The name of the relation model.

Many2Many.

**origin**

The name of the field that has the `Many2One` or Reference to the record.

Many2Many.

**target**

The name of the field that has the `Many2One` to the target record.

**Note:** A `Many2Many` field can be used on a simple `ModelView`, like in a `Wizard`. For this, `relation_name` is set to the target model and `origin` and `target` are set to `None`.

---

`Many2Many` has some extra optional arguments:

Many2Many.

**order**

Same as `One2Many.order`

Many2Many.

**datetime_field**

Same as `Many2One.datetime_field`

Many2Many.

**size**

An integer or a PYSON expression denoting the maximum number of records allowed in the relation.

Many2Many.

**add_remove**

An alias to the `domain` for compatibility with the `One2Many`.

Many2Many.

**filter**

Same as `One2Many.filter`

**Instance methods:**

Many2Many.

**get_target**()

Return the target `Model`. 

---

70 Chapter 6. Contents
**One2One**

```python
class trytond.model.fields.One2One(relation_name, origin, target, string[, datetime_field[, **options]]):
```

A one-to-one relation field.

**Warning:** It is on the relation_name Model that the unicity of the couple (origin, target) must be checked.

- **One2One.datetime_field**
  - Same as Many2One.datetime_field

- **One2One.filter**
  - Same as One2Many.filter

**Instance methods:**

- **One2One.get_target()**
  - Return the target Model.

**Function**

```python
class trytond.model.fields.Function(field, getter[, setter[, searcher]]):
```

A function field can emulate any other given field.

**Function** has a required argument:

- **Function.getter**
  - The name of the classmethod or instance of the Model for getting values. The signature of the classmethod is:

```python
getter(instances, name)
```

where `name` is the name of the field, and it must return a dictionary with a value for each instance.

Or the signature of the classmethod is:

```python
getter(instances, names)
```

where `names` is a list of name fields, and it must return a dictionary containing for each names a dictionary with a value for each instance.

The signature of the instancemethod is:

```python
getter(name)
```

where `name` is the name of the field, and it must return the value.

**Function** has some extra optional arguments:

- **Function.setter**
  - The name of the classmethod of the Model to set the value. The signature of the method id:

```python
setter(instances, name, value)
```

where `name` is the name of the field and `value` the value to set.
**Warning:** The modifications made to instances will not be saved automatically.

**Function.** `searcher`

The name of the classmethod of the `Model` to search on the field. The signature of the method is:

```
searcher(name, clause)
```

where `name` is the name of the field and `clause` is a domain clause. It must return a list of domain clauses but the operand can be a SQL query.

**Instance methods:**

**Function.** `get(ids, model, name[, values])`

Call the getter classmethod where `model` is the Model instance of the field, `name` is the name of the field.

**Function.** `set(ids, model, name, value)`

Call the setter classmethod where `model` is the Model instance of the field, `name` is the name of the field, `value` is the value to set.

**Function.** `search(model, name, clause)`

Call the searcher classmethod where `model` is the Model instance of the field, `name` is the name of the field, `clause` is a clause of domain.

**MultiValue**

```
class trytond.model.fields.MultiValue(field)
```

A multivalue field that is like a Function field but with predefined getter and setter that use the MultiValueMixin for stored values.

**Warning:** The `get_multivalue()` and `set_multivalue()` should be preferred over the descriptors of the field.

**Warning:** The default method of the field must accept pattern as keyword argument.

**Dict**

```
class trytond.model.fields.Dict(schema_model[, **options])
```

A dictionary field with predefined keys.

**Note:** It is possible to store the dict as JSON in the database if the backend supports by manually altering the column type to JSON on the database.

`Dict` has one extra required argument:

**Dict.** `schema_model`

The name of the DictSchemaMixin model that stores the definition of keys.

**Instance methods:**
Dict.translated([\textit{name}[\textit{type}]])

Returns a descriptor for the translated values or keys of the field following \textit{type}. The descriptor must be used on the same class as the field. Default \textit{type} is \textit{values}.

### 6.2.2 Wizard

A wizard is a finite state machine.

There is also a more practical introduction into wizards.

class trytond.wizard.Wizard(session_id)

This is the base for any wizard. It contains the engine for the finite state machine. A wizard must have some \textit{State} instance attributes that the engine will use.

Class attributes are:

Wizard.__name__

It contains the unique name to reference the wizard throughout the platform.

Wizard.start_state

It contains the name of the starting state.

Wizard.end_state

It contains the name of the ending state. If an instance method with this name exists on the wizard, it will be called on deletion of the wizard and it may return one of the client side action keywords.

Wizard.__rpc__

Same as \textit{trytond.model.Model.__rpc__}.

Wizard.states

It contains a dictionary with state name as key and \textit{State} as value.

Class methods are:

\textbf{classmethod} Wizard.__setup__()  
Setup the class before adding into the \textit{trytond.pool.Pool}.

\textbf{classmethod} Wizard.__post_setup__()  
Setup the class after added into the \textit{trytond.pool.Pool}.

\textbf{classmethod} Wizard.__register__(module_name)  
Register the wizard.

\textbf{classmethod} Wizard.create()  
Create a session for the wizard and returns a tuple containing the session id, the starting and ending state.

\textbf{classmethod} Wizard.delete(session_id)  
Delete the session.

\textbf{classmethod} Wizard.execute(session_id, data, state_name)  
Execute the wizard for the state. \textit{session_id} is a session id. \textit{data} is a dictionary with the session data to update. \textit{active_id}, \textit{active_ids}, \textit{active_model} and \textit{action_id} must be set in the context according to the records on which the wizard is run.

### 6.2.3 State

\textbf{class} trytond.wizard.State

This is the base for any wizard state.

Instance attributes are:
State.name
The name of the state.

6.2.4 StateView
class trytond.wizard.StateView(model_name, view, buttons)
A StateView is a state that will display a form in the client. The form is defined by the ModelView with the name model_name, the XML id in view and the buttons. The default value of the view can be set with a method on wizard having the same name as the state but starting with default_.

Instance attributes are:

   StateView.model_name
The name of the ModelView.

   StateView.view
The XML id of the form view.

   StateView.buttons
The list of Button instances to display on the form.

Instance methods are:

   StateView.get_view(wizard, state_name)
Returns the view definition like fields_view_get().

       • wizard is a Wizard instance
       • state_name is the name of the StateView instance

   StateView.get_defaults(wizard, state_name, fields)
Return default values for the fields.

       • wizard is a Wizard instance
       • state_name is the name of the State
       • fields is the list of field names

   StateView.get_buttons(wizard, state_name)
Returns button definitions of the wizard.

       • wizard is a Wizard instance
       • state_name is the name of the StateView instance

6.2.5 StateTransition
class trytond.wizard.StateTransition
A StateTransition brings the wizard to the state returned by the method having the same name as the state but starting with transition_.

6.2.6 StateAction
class trytond.wizard.StateAction(action_id)
A StateAction is a StateTransition which let the client launch an ir.action. This action definition can be customized with a method on wizard having the same name as the state but starting with do_.

Instance attributes are:
StateAction.action_id
The XML id of the ir.action.

Instance methods are:
StateAction.get_action()
Returns the ir.action definition.

6.2.7 StateReport

class trytond.wizard.StateReport(report_name)
A StateReport is a StateAction which find the report action by name instead of XML id.

6.2.8 Button

class trytond.wizard.Button(string, state[, icon[, default ]])
A Button is a single object containing the definition of a wizard button.

Instance attributes are:
Button.string
The label display on the button.

Button.state
The next state to reach if button is clicked.

Button.icon
The name of the icon to display on the button.

Button.default
A boolean to set it as default on the form.

6.2.9 PYSON

PYSON is the PYthon Statement and Object Notation.
There is also a more practical introduction into PYSON statements.

class trytond.pyson.PYSON
Base class of any PYSON statement. It is never used directly.

Instance methods:
PYSON.pyson()
Method that returns the internal dictionary representation of the statement.

PYSON.types()
Method that returns a set of all possible types which the statement can become when evaluated.

classmethod PYSON.eval(dct, context)
Method which returns the evaluation of the statement given in dct within the context. dct contains a dictionary which is the internal representation of a PYSON statement. context contains a dictionary with contextual values.
Encoder and Decoder

class trytond.pyson.PYSONEncoder
Encoder for PYSON statements into string representations.
Instance method:
PYSONEncoder.encode(object)
Returns a string representation of a given PYSON statement. object contains a PYSON statement.

class trytond.pyson.PYSONDecoder(context[, noeval])
Decoder for string into the evaluated or not PYSON statement.
Instance method:
PYSONDecoder.decode(object)
Returns a PYSON statement evaluated or not of a given string. object contains a string.

Statements

The following statements can be used in PYSON.

class trytond.pyson.Eval(value[, default])
An Eval() object represents the PYSON Eval() statement for evaluations. When evaluated, it returns the value of the statement named by value, if defined in the evaluation context, otherwise the default value (empty string by default). Returns an instance of itself.

Note: The default value determines the type of the statement.

class trytond.pyson.Not(value)
A Not object represents the PYSON Not() statement for logical negations. When evaluated, returns the boolean negation of the value of the statement named by value, if defined in the evaluation context. Returns an instance of itself.

class trytond.pyson.Bool(value)
A Bool object represents the PYSON Bool() statement for boolean evaluations. Returns the boolean representation of the value of the statement named by value.

class trytond.pyson.And(*statements)
An And object represents the PYSON And() statement for logical and operations. Returns the result of the logical conjunction of two or more values named by the statements in the statements tuple.

class trytond.pyson.Or(*statements)
An Or object represents the PYSON Or() statement for logical or operations. Returns the result of the logical disjunction of two or more values named by the statements in the statements tuple.

class trytond.pyson.Equal(statement1, statement2)
An Equal object represents the PYSON Equal() statement for equation comparisons. Returns true when a value of a statement named by statement1 and the value of a statement named by statement2 are equal, otherwise returns false.

class trytond.pyson.Greater(statement1, statement2[, equal])
A `Greater` object represents the PYSON `Greater()` statement for `greater-than` comparisons. Returns true when the value of the statement named by `statement1` is strictly greater than the value of the statement named by `statement2`, otherwise returns false. Is the value of the variable named by `equal` is true, then returns also true when both values of statements named by `statement1` and `statement2` are equal. In this case `Greater` works as a `greater-than or equal` operator.

**Note:** `None` value is replaced by `0` for the comparison.

```python
class trytond.pyson.Less(statement1, statement2[, equal])
```
A `Less` object represents the PYSON `Less()` statement for `less-than` comparisons. Returns true when the value of the statement named by `statement1` is strictly less than the value of the statement named by `statement2`, otherwise returns false. Is the value of the variable named `equal` is true, then returns also true when both values of the statements named by `statement1` and `statement2` are equal. In this case `Less` works as a `less-than or equal` operator.

**Note:** `None` value is replaced by `0` for the comparison.

```python
class trytond.pyson.If(condition, then_statement, else_statement)
```
An `If` object represents the PYSON `If()` statement for conditional flow control operations. Returns the value of the statement named by `then_statement` when the value of the statement named by `condition` evaluates true. Otherwise returns the value of the statement named by `else_statement`.

```python
class trytond.pyson.Get(obj, key[, default])
```
A `Get` object represents the PYSON `Get()` statement for dictionary look-up operations and evaluation. Look up and returns the value of a key named by `key` in an object named by `obj` if defined. Otherwise returns the value of the variable named by `default`.

```python
class trytond.pyson.In(key, obj)
```
An `In` object represents the PYSON `In()` statement for look-up dictionary or integer objects. Returns true when a list (or dictionary) object named by `obj` contains the value of the variable (or key) named by `key`. Otherwise returns false.

```python
class trytond.pyson.Date([year[, month[, day[, delta_years[, delta_month[, delta_days]]]]]])
```
A `Date` object represents the PYSON `Date()` statement for date related conversions and basic calculations. Returns a date object which represents the values of arguments named by the variables explained below. Missing values of arguments named by `year` or `month` or `day` take their defaults from the actual date. When values of arguments named by `delta_*` are given, they are added to the values of the appropriate arguments in a date and time preserving manner.

Arguments:

- `year` Contains a PYSON statement of type int or long.
- `month` Contains a PYSON statement of type int or long.
- `day` Contains a PYSON statement of type int or long.
- `delta_years` Contains a PYSON statement of type int or long.
- `delta_month` Contains a PYSON statement of type int or long.
- `delta_days` Contains a PYSON statement of type int or long.
A `DateTime` object represents the PYSON `Date()` statement for date and time related conversions and calculations. Returns a date time object which represents the values of variables named by the `arguments` explained below. Missing values of arguments named by `year`, `month`, `day`, `hour`, `minute`, `second`, `microseconds` take their defaults from the actual date and time. When values of arguments named by `delta_*` are given, these are added to the appropriate attributes in a date and time preserving manner.

Arguments:

- `year` Contains a PYSON statement of type int or long.
- `month` Contains a PYSON statement of type int or long.
- `day` Contains a PYSON statement of type int or long.
- `hour` Contains a PYSON statement of type int or long.
- `minute` Contains a PYSON statement of type int or long.
- `second` Contains a PYSON statement of type int or long.
- `microsecond` Contains a PYSON statement of type int or long.
- `delta_years` Contains a PYSON statement of type int or long.
- `delta_month` Contains a PYSON statement of type int or long.
- `delta_days` Contains a PYSON statement of type int or long.
- `delta_hours` Contains a PYSON statement of type int or long.
- `delta_minutes` Contains a PYSON statement of type int or long.
- `delta_seconds` Contains a PYSON statement of type int or long.
- `delta_microseconds` Contains a PYSON statement of type int or long.

A `Len` object represents the PYSON `Len()` statement for length of a dictionary, list or string. Returns the number of items in `value`.

A `Id` object represents the PYSON `Id()` statement for filesystem id evaluations. When converted into the internal dictionary, it returns the database id stored in `ir.model.data`.

### 6.2.10 Transaction

This class represents a Tryton transaction that contains thread-local parameters of a database connection. The Transaction instances are context manager that will commit or rollback the database transaction. In the event of an exception the transaction is rolled back, otherwise it is commited.

```python
class trytond.transaction.Transaction
```

- `database` The database.
- `readonly`
Transaction.connection
The database connection as defined by the PEP-0249.

Transaction.user
The id of the user.

Transaction.context

Transaction.create_records

Transaction.delete_records

Transaction.delete

Transaction.timestamp

Transaction.language
The language code defines in the context.

Transaction.counter
Count the number of modification made in this transaction.

Transaction.start(database_name, user[readonly][context[close, autocommit]])
Start a new transaction and return a context manager. The non-readonly transaction will be committed when exiting the with statement without exception. The other cases will be rollbacked.

Transaction.stop(commit)
Stop the transaction. If commit is True, the transaction will be committed otherwise it will be rollbacked. The context manager returned by Transaction.start() should be used instead of calling this method.

Transaction.set_context(context, **kwargs)
Update the transaction context and return a context manager. The context will be restored when exiting the with statement.

Transaction.reset_context()
Clear the transaction context and return a context manager. The context will be restored when exiting the with statement.

Transaction.set_user(user, set_context)
Modify the user of the transaction and return a context manager. set_context will put the previous user id in the context to simulate the record rules. The user will be restored when exiting the with statement.

Transaction.set_current_transaction(transaction)
Add a specific transaction on the top of the transaction stack. A transaction is committed or rollbacked only when its last reference is popped from the stack.

Transaction.new_transaction([autocommit, readonly])
Create a new transaction with the same database, user and context as the original transaction and adds it to the stack of transactions.

Transaction.commit()
Commit the transaction and all data managers associated.

Transaction.rollback()
Rollback the transaction and all data managers associated.

Transaction.join(datamanager)
Register in the transaction a data manager conforming to the Two-Phase Commit protocol. More information on how to implement such data manager is available at the Zope documentation. This method returns the registered datamanager. It could be a different yet equivalent (in term of python equality) datamanager than the one passed to the method.
Transaction\texttt{.atexit(func, *args, **kwargs)}

Register a function to be executed upon normal transaction termination. The function can not use the current transaction because it will be already committed or rollbaced.

### 6.2.11 Tools

Tools API reference.

#### Miscellaneous

\texttt{trytond.tools.resolve(name)}

Resolve a dotted name to a global object.

#### Singleton

\texttt{class trytond.tools.singleton.Singleton}

A class to use as metaclass to create a \texttt{singleton} object.

### 6.2.12 Pool

\texttt{class trytond.pool.Pool([database_name])}

The Pool store the instances of models, wizards and reports per database.

Static methods:

\texttt{static Pool.register(klass[, type])}

Register a class of type (default: \texttt{model}).

Class methods:

\texttt{classmethod Pool.start()}

Start the pool by registering all Tryton modules found.

\texttt{classmethod Pool.stop(database_name)}

Stop the pool by removing instances for the database.

\texttt{classmethod Pool.database_list()}

List all started database.

Instance methods:

\texttt{Pool.get(name[, type])}

Return the named instance of type from the pool.

\texttt{Pool.iterobject([type])}

Return an interator over instances names.

\texttt{Pool.fill(module)}

Fill the pool with the registered class from the module and return a list of classes for each type in a dictionary.

\texttt{Pool.setup([classes])}

Call all setup methods of the classes provided or for all the registered classes.
6.2.13 PoolMeta

class trytond.pool.PoolMeta

The PoolMeta is a metaclass helper to setup __name__ on class to be registered in the Pool.

6.2.14 PoolBase

class trytond.pool.PoolBase

The base class of registered class that will be setup.

6.2.15 RPC

class trytond.rpc.RPC([readonly, instantiate[, result[, check_access[, unique ]]]]])

RPC is an object to define the behavior of Remote Procedure Call.

Instance attributes are:

RPC.readonly
   The transaction mode

RPC.instantiate
   The position or the slice of the argument to be instanciated

RPC.result
   The function to transform the result

RPC.check_access
   Set _check_access in the context to activate the access right on model and field. Default is True.

RPC.unique
   If set, it ensures the instantiated records are unique. Default is True.

6.2.16 Sendmail

trytond.sendmail.sendmail_transactional(from_addr, to_addrs, msg[, transaction[, data-manager]])

Send email message only if the current transaction is successfully committed. The required arguments are an RFC 822 from-address string, a list of RFC 822 to-address strings (a bare string will be treated as a list with 1 address), and an email message. The caller may pass a Transaction instance to join otherwise the current one will be joined. A specific data manager can be specified otherwise the default SMTPDataManager will be used for sending email.

**Warning:** An SMTP failure will be only logged without raising any exception.

trytond.sendmail.sendmail(from_addr, to_addrs, msg[, server])

Send email message like sendmail_transactional() but directly without caring about the transaction. The caller may pass a server instance from smtplib.

trytond.sendmail.get_smtp_server([uri])

Return a SMTP instance from smtplib using the uri or the one defined in the email section of the configuration.

class trytond.sendmail.SMTPDataManager([uri])
SMTPDataManager implements a data manager which send queued email at commit. An option optional uri can be passed to configure the SMTP connection.

\[\text{SMTPDataManager}.\text{put} \text{ (from_addr, to_addrs, msg)}\]
Queue the email message to send.

### 6.2.17 FileStore

class trytond.filestore.FileStore
The class is used to store and retrieve files from the directory defined in the configuration path of database section. It uses a two levels of directory composed of the 2 chars of the file hash. It is an append only storage.

\[\text{trytond.filestore}.\text{get} \text{ (id[, prefix ])}\]
Retrieve the content of the file referred by the id in the prefixed directory.

\[\text{trytond.filestore}.\text{getmany} \text{ (ids[, prefix ])}\]
Retrieve a list of contents for the sequence of ids.

\[\text{trytond.filestore}.\text{size} \text{ (id[, prefix ])}\]
Return the size of the file referred by the id in the prefixed directory.

\[\text{trytond.filestore}.\text{sizemany} \text{ (ids[, prefix ])}\]
Return a list of sizes for the sequence of ids.

\[\text{trytond.filestore}.\text{set} \text{ (data[, prefix ])}\]
Store the data in the prefixed directory and return the identifiers.

\[\text{trytond.filestore}.\text{setmany} \text{ (data[, prefix ])}\]
Store the sequence of data and return a list of identifiers.

**Note:** The class can be overridden by setting a fully qualified name of a alternative class defined in the configuration class of the database section.

### 6.2.18 Cache

class trytond.cache.Cache (name[, size_limit[, context ]])
The class is used to cache values between server requests. The name should be unique and it’s used to identify the cache. We usually use <class_name>.<content_name> to make it unique. The size_limit field can be used to limit the number of values cached and the context parameter is used to indicate if the cache depends on the user context and is true by default. The cache is cleaned on Transaction starts and resets on Transaction commit or rollback.

**Warning:** As there is no deepcopy of the values cached, they must never be mutated after being set in or retrieved from the cache.

\[\text{trytond.cache}.\text{get} \text{ (key[, default ])}\]
Retrieve the value of the key in the cache. If a default is specified it will be returned when the key is missing otherwise it will return None.

\[\text{trytond.cache}.\text{set} \text{ (key, value)}\]
Sets the value of the key in the cache.

```
trytond.cache.clear()
```

Clears all the keys in the cache.

```
static trytond.cache.clean(dbname)
```

Clean the cache for database `dbname`

```
static trytond.cache.reset(dbname, name)
```

Reset the `name` cache for database `dbname`

```
static trytond.cache.resets(dbname)
```

Resets all the caches stored for database `dbname`

```
static trytond.cache.drop(dbname)
```

Drops all the caches for database `dbname`

---

**Note:** By default Tryton uses a MemoryCache, but this behaviour can be overridden by setting a fully qualified name of an alternative class defined in the configuration class of the cache section.

---

### 6.2.19 Test

```
trytond.tests.test_tryton.DB_NAME
```

The name of the database to use for testing. Its value is taken from the environment variable of the same name.

```
trytond.tests.test_tryton.USER
```

The user id used to test the transactions

```
trytond.tests.test_tryton.CONTEXT
```

The context used to test the transactions

```
trytond.tests.test_tryton.activate_module(name)
```

Activates the module `name` for the tested database.

In case database does not exist and the `DB_CACHE` environment variable is set then Tryton will restore a backup found in the directory pointed by `DB_CACHE`. Otherwise it will proceed to the creation of the database and the activation of the module.

### 6.2.20 ModuleTestCase

```
class trytond.tests.test_tryton.ModuleTestCase
```

A subclass of `unittest.TestCase` that tests a Tryton module. Some tests are included to ensure that the module works properly.

This class creates a temporary database with the module activated in `setUpClass` and drops it in the `tearDownClass` method.

```
ModuleTestCase.module
```

Name of the tested module.
6.2.21 Helpers

trytond.tests.test_tryton.with_transaction(user=1, context=None)

This function returns a decorator to run a test case inside a transaction. The transaction is rolled back and the cache cleared at the end of the test.

doctest helpers

trytond.tests.test_tryton.doctest_setup()

A function that prepares the run of the doctest by creating a database and dropping it beforehand if necessary. This function should be used as the setUp parameter.

Deprecated since version The: doctest_setup function should not be used anymore to set up DocFileSuite. New modules should use activate_modules() instead.

trytond.tests.test_tryton.doctest_teardown()

A function that cleans up after the run of the doctest by dropping the database. It should be used as tearDown parameter when creating a DocFileSuite.

trytond.tests.test_tryton.doctest_checker

A specialized doctest checker to ensure the Python 2/3 compatibility

trytond.tests.test_tryton.suite()

A function returning a subclass of unittest.TestSuite that will drop the database if it does not exist prior to the run of the tests.

6.2.22 Tests tools

trytond.tests.tools.activate_modules(modules)

This function is used in proteus doctests to activate a list of modules in the scenario.

trytond.tests.tools.set_user(user, config)

This function will set the user of the config proteus connection to user.
CHAPTER 7

Indices, glossary and tables

- genindex
- modindex
- search
t

tryton.cache, 82
tryton.filestore, 82
tryton.model, 51
tryton.model.fields, 60
tryton.pool, 80
tryton.pyson, 75
tryton.rpc, 81
tryton.sendmail, 81
tryton.tests.test_tryton, 83
tryton.tests.tools, 84
tryton.tools, 80
tryton.tools.singleton, 80
tryton.transaction, 78
tryton.wizard, 73
Symbols

__name__ (trytond.model.Model attribute), 51
__name__ (trytond.wizard.Wizard attribute), 73
__post_setup__() (trytond.model.Model class method), 51
__post_setup__() (trytond.wizard.Wizard class method), 73
__register__() (trytond.model.Model class method), 51
__register__() (trytond.wizard.Wizard class method), 73
__rpc__ (trytond.model.Model attribute), 51
__rpc__ (trytond.wizard.Wizard attribute), 73
__setup__() (trytond.model.Model class method), 51
__setup__() (trytond.wizard.Wizard class method), 73
__table__() (trytond.model.ModelSQL class method), 56
__table_history__() (trytond.model.ModelSQL class method), 56
_buttons (trytond.model.ModelView attribute), 52
_constraints (trytond.model.ModelStorage attribute), 53
_error_messages (trytond.model.Model attribute), 51
_history (trytond.model.ModelSQL attribute), 55
_order (trytond.model.ModelSQL attribute), 55
_order_name (trytond.model.ModelSQL attribute), 55
_rec_name (trytond.model.Model attribute), 51
_sql_constraints (trytond.model.ModelSQL attribute), 55
_sql_error_messages (trytond.model.ModelSQL attribute), 56
_table (trytond.model.ModelSQL attribute), 55
_transition_state (trytond.model.Workflow attribute), 57
_transitions (trytond.model.Workflow attribute), 58

A

action_id (trytond.wizard.StateAction attribute), 74
activate_module() (in module trytond.tests.test_tryton), 83
activate_modules() (in module trytond.tests.tools), 84
add_remove (trytond.model.fields.Many2Many attribute), 70
add_remove (trytond.model.fields.One2Many attribute), 69
And (class in trytond.pyson), 76
atexit() (trytond.transaction.Transaction method), 79
autocomplete (trytond.model.fields.Char attribute), 64

B

BigInteger (class in trytond.model.fields), 64
Binary (class in trytond.model.fields), 66
Bool (class in trytond.pyson), 76
Boolean (class in trytond.model.fields), 64
drop() (trytond.model.ModelStorage class method), 54
Button (class in trytond.wizard), 75
button() (trytond.model.ModelView static method), 52
button_action() (trytond.model.ModelView static method), 52
button_change() (trytond.model.ModelView static method), 52
buttons (trytond.wizard.StateView attribute), 74

C

Cache (class in trytond.cache), 82
Char (class in trytond.model.fields), 64
Check (class in trytond.model), 57
check_access (trytond.rpc.RPC attribute), 81
check_recursion() (trytond.model.ModelStorage class method), 55
check_xml_record() (trytond.model.ModelStorage class method), 55
trytond Documentation, Release 4.6

clean() (in module trytond.cache), 83
clear() (in module trytond.cache), 83
columns (trytond.model.Unique attribute), 57
commit() (trytond.transaction.Transaction method), 79
connection (trytond.transaction.Transaction attribute), 78
Constraint (class in trytond.model), 57
CONTEXT (in module trytond.tests.test_tryton), 83
count() (trytond.transaction.Transaction attribute), 79
counter (trytond.model.ModelStorage class method), 63
clear() (trytond.model.ModelStorage class method), 54
copy() (trytond.pool.Pool class method), 54
digits (trytond.model.ModelStorage class method), 54
digits (trytond.pool.Pool class method), 54
digits (trytond.pool.pool module), 54
digits (trytond.pool.pool), 54
digit_label (trytond.model.ModelStorage class method), 63
domains (trytond.model.ModelStorage class method), 63
default_create_uid() (trytond.model.ModelStorage static method), 54
default_create_date() (trytond.model.ModelStorage static method), 54
default_digits() (trytond.model.DictSchemaMixin static method), 59
default_get() (trytond.model.Model class method), 52
delete (trytond.transaction.Transaction attribute), 79
delete() (trytond.model.ModelStorage class method), 54
delete() (trytond.wizard.Wizard class method), 73
delete_records (trytond.transaction.Transaction attribute), 79
depends (trytond.model.ModelStorage class method), 62
depends() (in module trytond.model.fields), 64
Dict (class in trytond.model), 72
DictSchemaMixin (class in trytond.model), 58
digits (trytond.pool.pool), 54
digits (trytond.wizard.Wizard attribute), 73
digits (trytond.model.ModelStorage static method), 54
doctest_checker (in module trytond.tests.test_tryton), 84
doctest_setup() (in module trytond.tests.test_tryton), 84
doctest_teardown() (in module trytond.tests.test_tryton), 84
domain (trytond.model.ModelStorage class method), 61
drop() (in module trytond.cache), 83

database (trytond.transaction.Transaction attribute), 78
database_list() (trytond.pool.Pool class method), 80
Date (class in trytond.model.fields), 65
Date (class in trytond.pyson), 77
DateTime (class in trytond.model.fields), 66
DateTime (class in trytond.pyson), 77
datetime_field (trytond.model.fields.Many2Many attribute), 70
datetime_field (trytond.model.fields.Many2One attribute), 69
datetime_field (trytond.model.fields.One2Many attribute), 70
datetime_field (trytond.model.fields.One2One attribute), 71
datetime_field (trytond.model.fields.Reference attribute), 68
DB_NAME (in module trytond.tests.test_tryton), 83
decode() (trytond.pyson.PYSONDecoder method), 76
default() (trytond.wizard.Button attribute), 75
default_create_date() (trytond.model.ModelStorage static method), 54
default_create_uid() (trytond.model.ModelStorage static method), 54

declare() (trytond.model.ModelStorage class method), 52
decode() (trytond.pyson.PYSONDecoder method), 76
default() (trytond.wizard.Button attribute), 75
default_create_date() (trytond.model.ModelStorage static method), 54
default_create_uid() (trytond.model.ModelStorage static method), 54
default_digits() (trytond.model.DictSchemaMixin static method), 59
default_get() (trytond.model.Model class method), 52
delete (trytond.transaction.Transaction attribute), 79
delete() (trytond.model.ModelStorage class method), 54
delete() (trytond.wizard.Wizard class method), 73
delete_records (trytond.transaction.Transaction attribute), 79
depends (trytond.model.ModelStorage class method), 62
depends() (in module trytond.model.fields), 64
Dict (class in trytond.model), 72
DictSchemaMixin (class in trytond.model), 58
digits (trytond.pool.pool), 54
digits (trytond.model.ModelStorage class method), 54
digits (trytond.model.ModelStorage static method), 54
doctest_checker (in module trytond.tests.test_tryton), 84
doctest_setup() (in module trytond.tests.test_tryton), 84
doctest_teardown() (in module trytond.tests.test_tryton), 84
domain (trytond.model.ModelStorage class method), 61
drop() (in module trytond.cache), 83
ondelete (tryton.model.fields.Many2One attribute), 68
One2Many (class in tryton.model.fields), 69
One2One (class in tryton.model.fields), 71
Or (class in tryton.pyson), 76
order (tryton.model.fields.Many2Many attribute), 70
order (tryton.model.fields.One2Many attribute), 70
origin (tryton.model.fields.Many2Many attribute), 70

P
Pool (class in tryton.pool), 80
PoolBase (class in tryton.pool), 81
pre_validate() (tryton.model.Model class method), 51
put() (tryton.sendmail.SMTPDataManager method), 82
PYSON (class in tryton.pyson), 75
pyson() (tryton.pyson.PYSON method), 75
PYSONDecoder (class in tryton.pyson), 76

R
raise_user_error() (tryton.model.Model class method), 51
raise_user_warning() (tryton.model.Model class method), 51
read() (tryton.model.ModelStorage class method), 54
readonly (tryton.model.fields.Field attribute), 61
readonly (tryton.rpc.RPC attribute), 81
readonly (tryton.transaction.Transaction attribute), 78
rec_name (tryton.model.ModelStorage attribute), 53
Reference (class in tryton.model.fields), 68
register() (tryton.pool.Pool static method), 80
relation_name (tryton.model.fields.Many2Many attribute), 70
required (tryton.model.fields.Field attribute), 61
reset() (in module tryton.cache), 83
reset_context() (tryton.transaction.Transaction method), 79
resets() (in module tryton.cache), 83
resolve() (in module tryton.tools), 80
restore_history() (tryton.model.ModelSQL class method), 56
restore_history_before() (tryton.model.ModelSQL class method), 56
result (tryton.rpc.RPC attribute), 81
right (tryton.model.fields.Many2One attribute), 68
rollback() (tryton.transaction.Transaction method), 79
RPC (class in tryton.rpc), 81

S
save() (tryton.model.ModelStorage class method), 55
schema_model (tryton.model.fields.Dict attribute), 72
search() (tryton.model.fields.Function method), 72
search() (tryton.model.ModelSQL class method), 56
search() (tryton.model.ModelStorage class method), 54
search_count() (tryton.model.ModelStorage class method), 54
search_domain() (tryton.model.ModelSQL class method), 56
search_global() (tryton.model.ModelStorage class method), 54
search_read() (tryton.model.ModelStorage class method), 54
search_rec_name() (tryton.model.ModelStorage class method), 54
searcher (tryton.model.fields.Function attribute), 72
select (tryton.model.fields.Field attribute), 62
Selection (class in tryton.model.fields), 67
selection (tryton.model.fields.Reference attribute), 68
selection (tryton.model.fields.Selection attribute), 67
selection_change_with (tryton.model.fields.Reference attribute), 68
selection_change_with (tryton.model.fields.Selection attribute), 67
selection_json (tryton.model.DictSchemaMixin attribute), 59
selection_sorted (tryton.model.DictSchemaMixin attribute), 59
sendmail() (in module tryton.sendmail), 81
sendmail_transactional() (in module tryton.sendmail), 81
sequence_ordered() (in module tryton.model), 60
set() (in module tryton.cache), 82
set() (in module tryton.filestore), 82
set() (tryton.model.fields.Function method), 72
set_context() (tryton.transaction.Transaction method), 79
set_current_transaction() (tryton.transaction.Transaction method), 79
set_multivalue() (tryton.model.MultivalueMixin method), 60
set_rpc() (tryton.model.fields.Field method), 63
set_user() (in module tryton.tests.tools), 84
set_user() (tryton.transaction.Transaction method), 79
setmany() (in module tryton.filestore), 82
setter (tryton.model.fields.Function attribute), 71
setter_multivalue() (try-
view (trytond.wizard.StateView attribute), 74
view_attributes() (trytond.model.ModelView class method), 53
view_toolbar_get() (trytond.model.ModelView class method), 53

W
with_transaction() (in module trytond.tests.test_tryton), 84
Wizard (class in trytond.wizard), 73
Workflow (class in trytond.model), 57
write() (trytond.model.ModelStorage class method), 54
write_date (trytond.model.ModelStorage attribute), 53
write_uid (trytond.model.ModelStorage attribute), 53